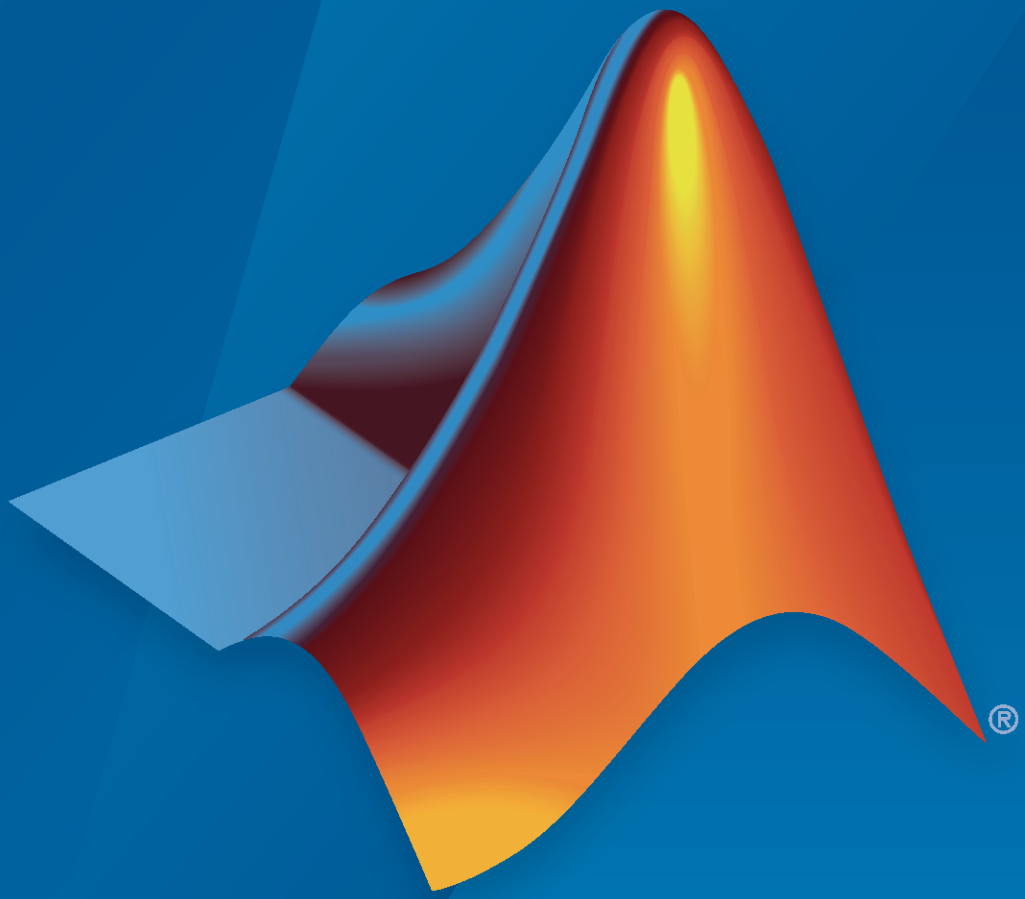
Datafeed Toolbox™

User's Guide



MATLAB®

R2022b

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Datafeed Toolbox™ User's Guide

Revision History

December 1999	First printing	New for MATLAB® 5.3 (Release 11)
June 2000	Online only	Revised for Version 1.2
December 2000	Online only	Revised for Version 1.3
February 2003	Online only	Revised for Version 1.4
June 2004	Online only	Revised for Version 1.5 (Release 14)
August 2004	Online only	Revised for Version 1.6 (Release 14+)
September 2005	Second printing	Revised for Version 1.7 (Release 14SP3)
March 2006	Online only	Revised for Version 1.8 (Release 2006a)
September 2006	Online only	Revised for Version 1.9 (Release 2006b)
March 2007	Third printing	Revised for Version 2.0 (Release 2007a)
September 2007	Online only	Revised for Version 3.0 (Release 2007b)
March 2008	Online only	Revised for Version 3.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.2 (Release 2008b)
March 2009	Online only	Revised for Version 3.3 (Release 2009a)
September 2009	Online only	Revised for Version 3.4 (Release 2009b)
March 2010	Online only	Revised for Version 3.5 (Release 2010a)
September 2010	Online only	Revised for Version 4.0 (Release 2010b)
April 2011	Online only	Revised for Version 4.1 (Release 2011a)
September 2011	Online only	Revised for Version 4.2 (Release 2011b)
March 2012	Online only	Revised for Version 4.3 (Release 2012a)
September 2012	Online only	Revised for Version 4.4 (Release 2012b)
March 2013	Online only	Revised for Version 4.5 (Release 2013a)
September 2013	Online only	Revised for Version 4.6 (Release 2013b)
March 2014	Online only	Revised for Version 4.7 (Release 2014a)
October 2014	Online only	Revised for Version 5.0 (Release 2014b)
March 2015	Online only	Revised for Version 5.1 (Release 2015a)
September 2015	Online only	Revised for Version 5.2 (Release 2015b)
March 2016	Online only	Revised for Version 5.3 (Release 2016a)
September 2016	Online only	Revised for Version 5.4 (Release 2016b)
March 2017	Online only	Revised for Version 5.5 (Release 2017a)
September 2017	Online only	Revised for Version 5.6 (Release 2017b)
March 2018	Online only	Revised for Version 5.7 (Release 2018a)
September 2018	Online only	Revised for Version 5.8 (Release 2018b)
March 2019	Online only	Revised for Version 5.8.1 (Release 2019a)
September 2019	Online only	Revised for Version 5.9 (Release 2019b)
March 2020	Online only	Revised for Version 5.9.1 (Release 2020a)
September 2020	Online only	Revised for Version 5.9.2 (Release 2020b)
March 2021	Online only	Revised for Version 6.0 (Release 2021a)
September 2021	Online only	Revised for Version 6.1 (Release 2021b)
March 2022	Online only	Revised for Version 6.2 (Release 2022a)
September 2022	Online only	Revised for Version 6.3 (Release 2022b)

Contents

1 Getting Started

Datafeed Toolbox Product Description 1-2

Data Server Connection Requirements 1-3
License Requirements 1-3
Proxy Information Requirements 1-4
Platform Requirements 1-4

Installing Bloomberg and Configuring Connections 1-5
Install Bloomberg Software 1-5
Add JAR Files to the MATLAB Java Class Path 1-5
Run Bloomberg Communications Server 1-6

Retrieve Current and Historical Data Using Bloomberg 1-7

Retrieve Historical Data Using FRED 1-10

Retrieve Historical Data Using Haver Analytics 1-12

Create Order Using X_TRADER 1-13

Workflows for Trading Technologies X_TRADER 1-16

Listen for X_TRADER Price Updates 1-18

Listen for X_TRADER Price Market Depth Updates 1-20

Submit X_TRADER Orders 1-23

Writing and Running Custom Event Handler Functions 1-26
Write Custom Event Handler Function 1-26
Run Custom Event Handler Function 1-26
Workflow for Custom Event Handler Function 1-27

2 Communicate with Financial Data Servers

Communicating with Data Service Providers 2-2

Comparing Bloomberg Connections 2-4

3 Data Provider Workflows

Connect to Bloomberg **3-2**

Retrieve Bloomberg Current Data **3-5**

Retrieve Bloomberg Historical Data **3-7**

Retrieve Bloomberg Intraday Tick Data **3-11**

Retrieve Bloomberg Real-Time Data **3-13**

Workflow for Bloomberg **3-15**
Bloomberg Desktop, Bloomberg Server, or Bloomberg B-PIPE Services . . **3-15**

Workflow for CQG **3-16**

Create Order Using CQG **3-18**

Create CQG Orders **3-20**

Request CQG Historical Data **3-24**

Request CQG Intraday Tick Data **3-27**

Request CQG Real-Time Data **3-30**

4 Bloomberg EMSX Topics

Create Order Using Bloomberg EMSX **4-2**

Create and Manage Bloomberg EMSX Order **4-4**

Create and Manage Bloomberg EMSX Route **4-8**

Manage Bloomberg EMSX Order and Route **4-12**

Workflow for Bloomberg EMSX **4-16**

5 Topics for Bloomberg C++ Interfaces

Create Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface **5-2**

Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface **5-4**

Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface **5-7**

Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface **5-11**

Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface **5-15**

Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface **5-18**

Retrieve Bloomberg Historical Data Using Bloomberg Desktop C++ Interface **5-20**

Retrieve Bloomberg Intraday Tick Data Using Bloomberg Desktop C++ Interface **5-24**

Retrieve Bloomberg Real-Time Data Using Bloomberg Desktop C++ Interface **5-26**

Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface **5-28**

Retrieve Bloomberg Historical Data Using Bloomberg B-PIPE C++ Interface **5-31**

Retrieve Bloomberg Intraday Tick Data Using Bloomberg B-PIPE C++ Interface **5-35**

Retrieve Bloomberg Real-Time Data Using Bloomberg B-PIPE C++ Interface **5-37**

Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface **5-39**

Retrieve Bloomberg Historical Data Using Bloomberg Server C++ Interface **5-41**

Retrieve Bloomberg Intraday Tick Data Using Bloomberg Server C++ Interface **5-45**

Retrieve Bloomberg Real-Time Data Using Bloomberg Server C++ Interface **5-47**

6 Transaction Cost Analysis

Analyze Trading Execution Results **6-2**

Post-Trade Analysis Metrics Definitions **6-5**
Implementation Shortfall **6-5**
Alpha Capture **6-5**
Benchmark Costs **6-5**
Broker Value Add **6-5**
Z-Score **6-6**

Kissell Research Group Data Sets **6-7**
Basket Variables **6-7**
BrokerNames Variables **6-7**
TradeData Variables **6-7**
TradeDataCurrent and TradeDataHistorical Variables **6-9**
PortfolioData Variables **6-9**
PostTradeData Variables **6-10**
TradeDataBackTest Variables **6-12**
TradeDataStressTest Variables **6-12**
TradeDataPortOpt Variables **6-13**
TradeDataTradeOpt Variables **6-14**
CovarianceData Table **6-15**
CovarianceTradeOpt Table **6-15**

Conduct Sensitivity Analysis to Estimate Trading Costs **6-17**

Estimate Portfolio Liquidation Costs **6-20**

Optimize Percentage of Volume Trading Strategy **6-23**

Optimize Trade Time Trading Strategy **6-26**

Optimize Trade Schedule Trading Strategy **6-29**

Estimate Trading Costs for Collection of Stocks **6-33**

Conduct Back Test on Portfolio **6-35**

Conduct Stress Test on Portfolio **6-38**

Liquidate Dollar Value from Portfolio **6-43**

Optimize Long Portfolio **6-48**

Determine Buy-Sell Imbalance Using Cost Index **6-51**

Rank Broker Performance **6-55**

Optimize Trade Schedule Trading Strategy for Basket **6-60**

Create Basket Summary and Efficient Trading Frontier **6-64**

7 Money.Net Web Socket Interface Topics

Retrieve Current and Historical Money.Net Data Using Money.Net Web Socket Interface **7-2**

Retrieve Real-Time Money.Net Data Using Money.Net Web Socket Interface **7-4**

Retrieve Money.Net News Stories Using Money.Net Web Socket Interface **7-6**

8 Money.Net Topics

Retrieve Current and Historical Money.Net Data **8-2**

Retrieve Real-Time Money.Net Data **8-5**

Retrieve Money.Net News Stories **8-7**

Money.Net Error and Warning Messages **8-10**
Money.Net Connection Error Messages **8-10**
Money.Net Data Retrieval Error Messages **8-10**
Money.Net Data Retrieval Warning Messages **8-11**

9 Twitter Topics

Conduct Sentiment Analysis Using Historical Tweets **9-2**

Tweet Based on Retrieved Twitter Data **9-6**

10 Tick History from Refinitiv Topics

Decide to Buy Shares with Intraday Data Using Tick History from Refinitiv **10-2**

Decide to Sell Shares with Historical Data Using Tick History from Refinitiv **10-4**

11 Quandl Topics

Access Quandl Error Messages **11-2**

12 Datastream Web Services Topics

Retrieve Datastream Web Services Historical Data **12-2**

Access Datastream Web Services Error Messages **12-4**

13 IHS Markit Topics

Retrieve Factor Rank Data for Portfolio Selection **13-2**

IHS Markit Error Messages **13-4**

14 WDS Topics

Decide to Buy Shares Using Current and Historical WDS Data **14-2**

Create Order Using Real-Time Snapshot WDS Data **14-4**

15 Functions

Getting Started

- “Datafeed Toolbox Product Description” on page 1-2
- “Data Server Connection Requirements” on page 1-3
- “Installing Bloomberg and Configuring Connections” on page 1-5
- “Retrieve Current and Historical Data Using Bloomberg” on page 1-7
- “Retrieve Historical Data Using FRED” on page 1-10
- “Retrieve Historical Data Using Haver Analytics” on page 1-12
- “Create Order Using X_TRADER” on page 1-13
- “Workflows for Trading Technologies X_TRADER” on page 1-16
- “Listen for X_TRADER Price Updates” on page 1-18
- “Listen for X_TRADER Price Market Depth Updates” on page 1-20
- “Submit X_TRADER Orders” on page 1-23
- “Writing and Running Custom Event Handler Functions” on page 1-26

Datafeed Toolbox Product Description

Access financial data from data service providers

Datafeed Toolbox provides access to financial data, news and social media data, and trading systems. You can establish connections from MATLAB® to retrieve historical, intraday, or real-time data streams and then perform analyses, develop models and financial trading strategies, and create visualizations that reflect financial and market behavior.

You can use the streaming and event-based data in MATLAB to build automated trading strategies that react to market events via industry-standard or proprietary trade execution platforms. The toolbox includes functions for analyzing transaction costs, accessing trade and quote pricing data, defining order types, and executing orders.

Supported data providers include Bloomberg®, FRED®, Haver Analytics®, Quandl®, Twitter®, and Refinitiv™. Supported trade execution platforms include Bloomberg EMSX, Trading Technologies® X_TRADER®, Wind Data Feed Services (WDS), and CQG®.

Data Server Connection Requirements

License Requirements

To connect to a data service provider, you must have a valid license for the required client software on your machine. For details, contact your data service sales representative or go to the provider website. For the list of websites, see “Communicating with Data Service Providers” on page 2-2. The following data service providers require you to install proprietary software on your computer or require you to obtain credentials.

- Bloomberg
  - Requires Bloomberg Desktop, Server, or B-PIPE® software license.
  - To install Bloomberg EMSX from Bloomberg L.P., find the latest installation files at `https://www.bloomberg.com`. You need a Bloomberg license to install and run Bloomberg EMSX.
- CQG
  - To install CQG, find the latest installation files at `https://www.cqg.com`. You need a CQG license to install and run CQG.
  - The Datafeed Toolbox no longer supports connection using a 32-bit version of MATLAB. To configure CQG to work with a 64-bit version of MATLAB, see the instructions in MATLAB Answers.
- FactSet®
  - Requires a license to use FactSet Workstation.
- Haver Analytics
  - Install the Haver Analytics software.
- Refinitiv data servers
  - To connect from the Internet to the Datastream™ Web Services from Refinitiv API, request a user name, password, and URL from Refinitiv.
- Trading Technologies
  - To install Trading Technologies, find the latest installation files at `https://www.tradingtechnologies.com`. You need a Trading Technologies license to install and run Trading Technologies.
- Twitter
  - To establish a Twitter connection, you must obtain these required credentials from Twitter:
    - Consumer key
    - Consumer secret
    - Access token
    - Access token secret

    To obtain these credentials, you must first log in to your Twitter account. Then, complete the form in Create an application.
- Wind Data Feed Services (WDS)

- To install Wind Data Feed Services (WDS) from The Wind Information Co., Ltd., find the latest installation files at `https://www.wind.com.cn/NewSite/data.html`.

See the MathWorks® website for the system requirements for connecting to these providers.

Proxy Information Requirements

If your network requires proxy authentication, these data service providers can require specification of a proxy host, proxy port, user name, and password:

- Datastream Web Services from Refinitiv
- FactSet
- FRED
- IHS Markit®
- Quandl
- Tick History from Refinitiv

For details, see “Specify Proxy Server Settings for Connecting to the Internet”.

Platform Requirements

These data service providers work only with the Windows® platform:

- Bloomberg
- CQG
- Haver Analytics
- Trading Technologies X_TRADER
- WDS

See Also

More About

- “Communicating with Data Service Providers” on page 2-2
- “Installing Bloomberg and Configuring Connections” on page 1-5

Installing Bloomberg and Configuring Connections

Datafeed Toolbox provides various ways to connect to Bloomberg by using Bloomberg interfaces and Bloomberg C++ interfaces. For details, see “Comparing Bloomberg Connections” on page 2-4. Before connecting to the Bloomberg interfaces, follow these required steps. For Bloomberg C++ interfaces, the only required step is to install the Bloomberg software.

Install Bloomberg Software

For the latest Bloomberg software, see `https://www.bloomberg.com`.

After installing the Bloomberg software, add Java® archive (JAR) files to the MATLAB Java class path and run the Bloomberg Communications Server according to these requirements.

Step	Bloomberg Desktop	Bloomberg Server	Bloomberg B-PIPE
Add `blpapi3.jar` JAR file to the MATLAB Java class path	Required for connection	Required for connection	Required for connection
Run the Bloomberg Communications Server	Required for connection	Required for connection	Not required

Add JAR Files to the MATLAB Java Class Path

Bloomberg Desktop, Bloomberg Server, and Bloomberg B-PIPE

With the Bloomberg V3 release, install the JAR file `blpapi3.jar` from Bloomberg. This JAR file ensures that `blp`, `blpsrv`, `bpipe`, and other Bloomberg commands work correctly.

If you have already downloaded `blpapi3.jar`, find it in a folder such as `c:\blp\DAPI\blpapi3.jar` or search for it in the Bloomberg installation folder `c:\blp`. When you have found `blpapi3.jar`, go to step 3.

Note For each Bloomberg connection, the folder location of the downloaded JAR file can be different. For questions, contact Bloomberg.

If you have not downloaded `blpapi3.jar` from Bloomberg, download it as follows:

1. In the Bloomberg terminal, type `WAPI <GO>` to open the API Developer’s Help Site screen.
2. Click API Download Center. Download and install the appropriate software.
3. Once you have `blpapi3.jar` on the system, add it to the MATLAB Java class path. There are two ways to add the JAR file:

   - Add the JAR file to the MATLAB Java class path for every MATLAB session using `javaaddpath`.

     ```
     javaaddpath c:\blp\DAPI\blpapi3.jar
     ```

     The JAR file path varies depending on the installed Bloomberg software.
   - Or, to automate adding this file, add `javaaddpath` to the `startup.m` file or add the full path for the JAR file to the `javaclasspath.txt` file.

To decide which way is best for you, see “Startup Options in MATLAB Startup File” and “Static Path of Java Class Path”.

4 If you modify `startup.m` or `javaclasspath.txt` files, restart MATLAB.

Troubleshooting Adding JAR Files

If the JAR files are missing from the MATLAB Java class path, this error message displays: `Please verify that blpapi3.jar is included on MATLAB javaclasspath`.

To add the JAR files to the MATLAB Java class path, follow the preceding instructions.

For issues with downloading and installing JAR files, contact Bloomberg.

Run Bloomberg Communications Server

The Bloomberg Communications Server is a program named `bbcomm.exe`. You can find this program in a folder such as `c:\blp\DAPI\bbcomm.exe` or search for it in the Bloomberg installation folder `c:\blp`. If `bbcomm.exe` does not start automatically, double-click it. An MS-DOS® command window opens and stays open while `bbcomm.exe` is running.

To avoid manually starting `bbcomm.exe` each time you turn on the computer, add `bbcomm.exe` to the startup folder:

1 Create a Windows shortcut to `bbcomm.exe`.

2 Move this shortcut to the startup folder. An example startup folder name is `C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup`.

When you are finished with the Bloomberg connection, stop `bbcomm.exe` by running `bbstop.exe`. You can find `bbstop.exe` in the same folder as `bbcomm.exe`. Double-click `bbstop.exe`. The MS-DOS command window closes.

See Also

`blp` | `blpsrv` | `bpipe`

Related Examples

- “Connect to Bloomberg” on page 3-2

More About

- “Data Server Connection Requirements” on page 1-3
- “Communicating with Data Service Providers” on page 2-2
- “Workflow for Bloomberg” on page 3-15

Retrieve Current and Historical Data Using Bloomberg

This example shows how to connect to Bloomberg® and retrieve current and historical Bloomberg® market data. For details about Bloomberg® connection requirements, see “Data Server Connection Requirements” on page 1-3. To ensure a successful Bloomberg connection, perform the required steps before executing a connection function. For details, see “Installing Bloomberg and Configuring Connections” on page 1-5.

Connect to Bloomberg®

Create a Bloomberg® Desktop connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg® Server using `blpsrv` or Bloomberg® B-PIPE® using `bpipe`.

Retrieve Current Data

Format MATLAB® data display for currency.

```
format bank
```

Retrieve closing and open prices for Microsoft®.

```
sec = 'MSFT US Equity';
fields = {'LAST_PRICE';'OPEN'}; % closing and open prices

[d,sec] = getdata(c,sec,fields)
```

```
d =

  struct with fields:

    LAST_PRICE: 62.32
          OPEN: 62.48


sec =

  cell

    'MSFT US Equity'
```

`d` contains the Bloomberg® closing and open prices. `sec` contains the Bloomberg® security name for Microsoft®.

Retrieve Historical Data

Retrieve monthly closing and open price data from January 1, 2012 through December 31, 2012 for Microsoft®.

```
fromdate = '1/01/2012'; % beginning of date range for historical data
todate = '12/31/2012'; % ending of date range for historical data
```

```
period = 'monthly'; % retrieve monthly data

[d,sec] = history(c,sec,fields,fromdate,todate,period)


d =

     734899.00         29.53         26.55
     734928.00         31.74         29.79
     734959.00         32.26         31.93
     734989.00         32.02         32.22
     735020.00         29.19         32.05
     735050.00         30.59         28.76
     735081.00         29.47         30.62
     735112.00         30.82         29.59
     735142.00         29.76         30.45
     735173.00         28.54         29.81
     735203.00         26.61         28.84
     735234.00         26.71         26.78


sec =

  cell

    'MSFT US Equity'
```

`d` contains the numeric representation of the date in the first column, closing price in the second column, and open price in the third column. Each row represents data for one month in the date range. `sec` contains the Bloomberg® security name for Microsoft®.

Find Maximum Open Price in Date Range

Calculate the maximum open price for the year 2012.

```
openprices = d(:,3); % retrieve all open prices in date range
max(openprices) % calculate maximum open price


ans =

         32.22
```

Close Bloomberg® Connection

```
close(c)
```

See Also

`blp` | `close` | `getdata` | `history`

Related Examples

- “Connect to Bloomberg” on page 3-2
- “Retrieve Bloomberg Current Data” on page 3-5

- “Retrieve Bloomberg Historical Data” on page 3-7
- “Retrieve Bloomberg Intraday Tick Data” on page 3-11
- “Retrieve Bloomberg Real-Time Data” on page 3-13

Retrieve Historical Data Using FRED

This example shows how to connect to FRED®, retrieve historical foreign exchange rates, and determine when the highest rate occurred.

Create FRED Connection

Connect to the FRED data server using the URL `'https://fred.stlouisfed.org/'`.

```
url = 'https://fred.stlouisfed.org/';
c = fred(url);
```

Retrieve Historical Foreign Exchange Rates

Adjust the display data format for currency.

```
format bank
```

Retrieve all historical data for the US / Euro Foreign Exchange Rate series. `d` contains the series description.

```
series = 'DEXUSEU';
```

```
d = fetch(c,series)
```

```
d = struct with fields:
                 Title: ' U.S. / Euro Foreign Exchange Rate'
              SeriesID: ' DEXUSEU'
                Source: ' Board of Governors of the Federal Reserve System (US)'
               Release: ' H.10 Foreign Exchange Rates'
    SeasonalAdjustment: ' Not Seasonally Adjusted'
             Frequency: ' Daily'
                 Units: ' U.S. Dollars to One Euro'
             DateRange: ' 1999-01-04 to 2018-06-15'
           LastUpdated: ' 2018-06-18 3:51 PM CDT'
                 Notes: ' Noon buying rates in New York City for cable transfers payable in forei
                  Data: [5075×2 double]
```

Display the numeric representation of the date and the foreign exchange rate for the first three rows of data.

```
d.Data(1:3,:)
```

```
ans = 3×2

     730124.00          1.18
     730125.00          1.18
     730126.00          1.16
```

Retrieve Historical Foreign Exchange Rates Using Date Range

Retrieve historical data from January 1 through June 1, 2012, for the US / Euro Foreign Exchange Rate series.

```
startdate = '01/01/2012'; % beginning of date range for historical data
enddate = '06/01/2012'; % ending of date range for historical data
```

```
d = fetch(c,series,startdate,enddate)
```

```
d = struct with fields:
                 Title: ' U.S. / Euro Foreign Exchange Rate'
              SeriesID: ' DEXUSEU'
                Source: ' Board of Governors of the Federal Reserve System (US)'
               Release: ' H.10 Foreign Exchange Rates'
    SeasonalAdjustment: ' Not Seasonally Adjusted'
             Frequency: ' Daily'
                 Units: ' U.S. Dollars to One Euro'
             DateRange: ' 1999-01-04 to 2018-06-15'
           LastUpdated: ' 2018-06-18 3:51 PM CDT'
                 Notes: ' Noon buying rates in New York City for cable transfers payable in forei
                  Data: [110×2 double]
```

Determine Highest Foreign Exchange Rate in Date Range

Determine the highest foreign exchange rate `maxforex` in the date range. `forex` contains all the exchange rates in the data.

```
forex = d.Data(:,2);
maxforex = max(forex)
```

```
maxforex =
          1.35
```

Determine when the highest foreign exchange rate occurred. To find the index `idx` for the highest foreign exchange rate, the function `find` uses the tolerance `value`. Retrieve the serial date number by indexing into the array of data using `idx`. Convert the serial date number to a character vector using the `datestr` function.

```
value = abs(forex-maxforex);
idx = find(value<0.001,1);
date = d.Data(idx,1);
datestr(date)
```

```
ans =
'24-Feb-2012'
```

Close FRED Connection

```
close(c)
```

See Also

`fred` | `close` | `fetch` | `datestr` | `find`

Retrieve Historical Data Using Haver Analytics

This example shows how to connect to Haver Analytics and retrieve historical data.

Connect to Haver Analytics

Connect to Haver Analytics using a daily file.

```
c = haver('c:\work\haver\haverd.dat');
```

Retrieve All Historical Data

Retrieve all historical data for the Haver Analytics variable FFED. The descriptor for this variable is Federal Funds [Effective] Rate (% p.a.).

```
variable = 'FFED'; % return data for FFED

d = fetch(c,variable);
```

Display the first three rows of data.

```
d(1:3,:)

ans =

     715511.00          2.38
     715512.00          2.50
     715515.00          2.50
```

d contains the numeric representation of the date and the closing value.

Retrieve Historical Data for a Date Range

Retrieve historical data from January 1, 2005, through December 31, 2005, for FFED.

```
fromdate = '01/01/2005'; % beginning of date range for historical data
todate = '12/31/2005'; % ending of date range for historical data

d = fetch(c,variable,fromdate,todate);
```

Close the Haver Analytics Connection

```
close(c)
```

Open the Haver Analytics User Interface

Use the `havertool` function to open the Haver Analytics User Interface. You can observe different Haver Analytics variables in a chart format.

```
c = haver('c:\work\haver\haverd.dat');

havertool(c)
```

For details, see the `havertool` function.

See Also

`haver` | `close` | `fetch` | `havertool`

Create Order Using X_TRADER

This example shows how to connect to Trading Technologies X_TRADER and create a market order.

Connect to Trading Technologies X_TRADER

```
c = xtrdr;
```

Create Instrument for Contract

Create an instrument for a contract of CAISO NP15 EZ Gen Hub 5 MW Peak Calendar-Day Real-Time LMP Futures with an expiration date of August 2014 on the Chicago Mercantile Exchange.

```
createInstrument(c,'Exchange','CME','Product','2F',...
                 'ProdType','Future','Contract','Aug14',...
                 'Alias','SubmitOrderInstrument3')
```

Register Event Handler for Order Server

Register an event handler to check the order server status.

```
sExchange = c.Instrument.Exchange;
c.Gate.registerevent({'OnExchangeStateUpdate', ...
    @(varargin)ttorderserverstatus(varargin{:},sExchange)})
```

Create Order Set and Set Order Properties

Create an empty order set. Then, set order set properties. Setting the first property to true (1) enables the X_TRADER API to send order rejection notifications. Setting the second property to true (1) enables the X_TRADER API to add order pairs for all order updates to the order tracker list in this order set. Setting the third property to `ORD_NOTIFY_NORMAL` sets the X_TRADER API notification mode for order status events to normal.

```
createOrderSet(c)

c.OrderSet(1).EnableOrderRejectData = 1;
c.OrderSet(1).EnableOrderUpdateData = 1;
c.OrderSet(1).OrderStatusNotifyMode = 'ORD_NOTIFY_NORMAL';
```

Set Position Limit Checks

```
c.OrderSet(1).Set('NetLimits',false)
```

Register Event Handlers for Order Status

Register event handlers to track events associated with the order status.

```
registerevent(c.OrderSet(1),{'OnOrderFilled',...
                             @(varargin)ttorderevent(varargin{:},c)})
registerevent(c.OrderSet(1),{'OnOrderRejected',...
                             @(varargin)ttorderevent(varargin{:},c)})
registerevent(c.OrderSet(1),{'OnOrderSubmitted',...
                             @(varargin)ttorderevent(varargin{:},c)})
registerevent(c.OrderSet(1),{'OnOrderDeleted',...
                             @(varargin)ttorderevent(varargin{:},c)})
```

Enable Order Submission

Open the instrument for trading and enable the X_TRADER API to retrieve market depth information when opening the instrument.

```
c.OrderSet(1).Open(1)
```

Build Order Profile with Existing Instrument

```
orderProfile = createOrderProfile(c,'Instrument',c.Instrument(1));
```

Set Customer Default Property

Assign the customer defaults for trading an instrument.

```
orderProfile.Customer = '<Default>';
```

Set Up Order Profile as Market Order

Set up the order profile as a market order for buying 225 shares.

```
orderProfile.Set('BuySell','Buy')
orderProfile.Set('Qty','225')
orderProfile.Set('OrderType','M')
```

Check Order Server Status

```
nCounter = 1;
while ~exist('bServerUp','var') && nCounter < 20
  % bServerUp is created by ttorderserverstatus
  pause(1)
  nCounter = nCounter + 1;
end
```

Verify Order Server Availability and Submit Order

```
if exist('bServerUp','var') && bServerUp
  % Submit the order
  submittedQuantity = c.OrderSet(1).SendOrder(orderProfile);
  disp(['Quantity Sent: '  num2str(submittedQuantity)])
else
  disp('Order server is down.  Unable to submit order.')
end
```

The X_TRADER API submits the order to the exchange and returns the number of contracts sent for lot-based contracts or the flow quantity sent for flow-based contracts in the output argument `submittedQuantity`.

Close Trading Technologies X_TRADER Connection

```
close(c)
```

See Also

`xtrdr` | `createInstrument` | `createOrderSet` | `createOrderProfile` | `close`

Related Examples

- “Listen for X_TRADER Price Updates” on page 1-18
- “Listen for X_TRADER Price Market Depth Updates” on page 1-20
- “Submit X_TRADER Orders” on page 1-23

More About

- “Workflows for Trading Technologies X_TRADER” on page 1-16

External Websites

- X_TRADER API Resources

Workflows for Trading Technologies X_TRADER

You can use X_TRADER to monitor market price information and submit orders.

To monitor market price information:

1. Connect to Trading Technologies X_TRADER using `xtrdr`.
2. Create an event notifier using `createNotifier`.
3. Create an instrument and attach it to the notifier using `createInstrument`. Optionally, use `getData` to return information on the instrument that you have created.
4. Close the Trading Technologies X_TRADER connection using `close`.

To submit orders to X_TRADER:

1. Connect to Trading Technologies X_TRADER using `xtrdr`.
2. Create an event notifier using `createNotifier`.
3. Create an instrument and attach it to the notifier using `createInstrument`. Optionally, use `getData` to return information on the instrument that you have created.
4. Create an order set using `createOrderSet` to define the level of the order status events and event handlers for orders that will be submitted to X_TRADER.
5. Define the order using `createOrderProfile`. An order profile contains the settings that define an individual order to be submitted.
6. Route the order for execution using the `OrderSet` object created by `createOrderSet` in step 4.
7. Close the Trading Technologies X_TRADER connection using `close`.

To monitor market price information and respond to market changes by automatically submitting orders to X_TRADER:

1. Connect to Trading Technologies X_TRADER using `xtrdr`.
2. Create an event notifier using `createNotifier`.
3. Create an instrument and attach it to the notifier using `createInstrument`. Use `getData` to return information on the instrument that you have created.
4. Define events by assigning callbacks for validating or invalidating an instrument and performing calculations based on the event. Based on some predefined condition reached when changes in the incoming data satisfy the condition, event callbacks execute the functions in steps 5, 6, and 7.
5. Create an order set using `createOrderSet` to define the level of the order status events and event handlers for orders that will be submitted to X_TRADER.
6. Define the order using `createOrderProfile`. An order profile contains the settings that define an individual order to be submitted.
7. Route the order for execution using the `OrderSet` object created by `createOrderSet` in step 5.
8. Close the Trading Technologies X_TRADER connection using `close`.

See Also

Related Examples

- “Create Order Using X_TRADER” on page 1-13

- “Listen for X_TRADER Price Updates” on page 1-18
- “Listen for X_TRADER Price Market Depth Updates” on page 1-20
- “Submit X_TRADER Orders” on page 1-23

Listen for X_TRADER Price Updates

This example shows how to connect to X_TRADER and listen for price update event data.

Connect to X_TRADER

```
X = xtrdr;
```

Create an Event Notifier

The event notifier is the X_TRADER mechanism that lets you define MATLAB functions to use as callbacks for specific events.

```
createNotifier(X)
```

Create an Instrument

Create an instrument and attach it to the notifier.

```
createInstrument(X,'Exchange','CME','Product','2F',...
                'ProdType','Future','Contract','Aug13',...
                'Alias','PriceInstrument1')
X.InstrNotify(1).AttachInstrument(X.Instrument(1))
```

Define Events

Assign callbacks for validating or invalidating an instrument, and for handling data updates for a previously validated instrument.

```
registerevent(X.InstrNotify(1),{'OnNotifyFound',...
                      @(varargin)ttinstrumentfound(varargin{:})})
registerevent(X.InstrNotify(1),{'OnNotifyNotFound',...
                      @(varargin)ttinstrumentnotfound(varargin{:})})
registerevent(X.InstrNotify(1),{'OnNotifyUpdate',...
                      @(varargin)ttinstrumentupdate(varargin{:})})
```

Monitor Events

Set the update filter to monitor the desired fields. In this example, events are monitored for updates to last price, last quantity, previous last quantity, and a change in prices. Listen for this event data.

```
X.InstrNotify(1).UpdateFilter = 'Last$,LastQty$,~LastQty$,Change$';
X.Instrument(1).Open(0)
```

The last command tells X_TRADER to start monitoring the attached instruments using the specified event settings.

Close the Connection

```
close(X)
```

See Also

`xtrdr` | `close` | `createInstrument` | `createNotifier`

Related Examples

- “Create Order Using X_TRADER” on page 1-13
- “Listen for X_TRADER Price Market Depth Updates” on page 1-20
- “Submit X_TRADER Orders” on page 1-23

More About

- “Workflows for Trading Technologies X_TRADER” on page 1-16

Listen for X_TRADER Price Market Depth Updates

This example shows how to connect to X_TRADER and turn on event handling for level-two market data (for example, bid and ask orders in the market for an instrument) and then create a figure window to display the depth data.

Connect to X_TRADER

```
X = xtrdr;
```

Create an Event Notifier

Create an event notifier and enable depth updates. The event notifier is the X_TRADER mechanism lets you define MATLAB functions to use as callbacks for specific events.

```
createNotifier(X)
X.InstrNotify(1).EnableDepthUpdates = 1;
```

Create an Instrument

```
createInstrument(X,'Exchange','CME','Product','2F','ProdType','Future',...
              'Contract','Aug13','Alias','PriceInstrumentDepthUpdate')
```

Attach an Instrument to a Notifier

Assign one or more notifiers to an instrument. A notifier can have one or more instruments attached to it.

```
X.InstrNotify(1).AttachInstrument(X.Instrument(1))
```

Define Events

Assign callbacks for validating or invalidating an instrument, and updating the example order book window.

```
registerevent(X.InstrNotify(1),{'OnNotifyFound',...
                                @ttinstrumentfound})
registerevent(X.InstrNotify(1),{'OnNotifyNotFound',...
                                @ttinstrumentnotfound})
registerevent(X.InstrNotify(1),{'OnNotifyDepthData',...
                                @ttinstrumentdepthupdate})
```

Set Up the Figure Window

Set up the figure window to display depth data.

```
f = figure('Numbertitle','off','Tag','TTPriceUpdateDepthFigure',...
       'Name',['Order Book - ' X.Instrument(1).Alias])
pos = f.Position;
f.Position = [pos(1) pos(2) 360 315];
f.Resize = 'off';
```

Create Controls

Create controls for the last price data.

```
bspc = 5;
bwid = 80;
```

```
bhgt = 20;

uicontrol('Style','text','String','Exchange',...
          'Position',[bspc 4*bspc+3*bhgt bwid bhgt])
uicontrol('Style','text','String','Product',...
          'Position',[2*bspc+bwid 4*bspc+3*bhgt bwid bhgt])
uicontrol('Style','text','String','Type',...
          'Position',[3*bspc+2*bwid 4*bspc+3*bhgt bwid bhgt])
uicontrol('Style','text','String','Contract',...
          'Position',[4*bspc+3*bwid 4*bspc+3*bhgt bwid bhgt])
ui.Exchange = uicontrol('Style','text','Tag','',...
          'Position',[bspc 3*bspc+2*bhgt bwid bhgt]);
ui.Product = uicontrol('Style','text','Tag','',...
          'Position',[2*bspc+bwid 3*bspc+2*bhgt bwid bhgt]);
ui.Type = uicontrol('Style','text','Tag','',...
          'Position',[3*bspc+2*bwid 3*bspc+2*bhgt bwid bhgt]);
ui.Contract = uicontrol('Style','text','Tag','',...
          'Position',[4*bspc+3*bwid 3*bspc+2*bhgt bwid bhgt]);
uicontrol('Style','text','String','Last Price',...
          'Position',[bspc 2*bspc+bhgt bwid bhgt])
uicontrol('Style','text','String','Last Qty',...
          'Position',[2*bspc+bwid 2*bspc+bhgt bwid bhgt])
uicontrol('Style','text','String','Change',...
          'Position',[3*bspc+2*bwid 2*bspc+bhgt bwid bhgt])
ui.Last = uicontrol('Style','text','Tag','',...
          'Position',[bspc bspc bwid bhgt]);
ui.Quantity = uicontrol('Style','text','Tag','',...
          'Position',[2*bspc+bwid bspc bwid bhgt]);
ui.Change = uicontrol('Style','text','Tag','',...
          'Position',[3*bspc+2*bwid bspc bwid bhgt]);
```

Create a Table

Create a table containing order information.

```
data = {' '};
data = data(ones(10,4));
uibook = uitable('Data',data,'ColumnName',...
                {'Bid','Bid Size','Ask','Ask Size'},...
                 'Position',[5 105 350 205]);
```

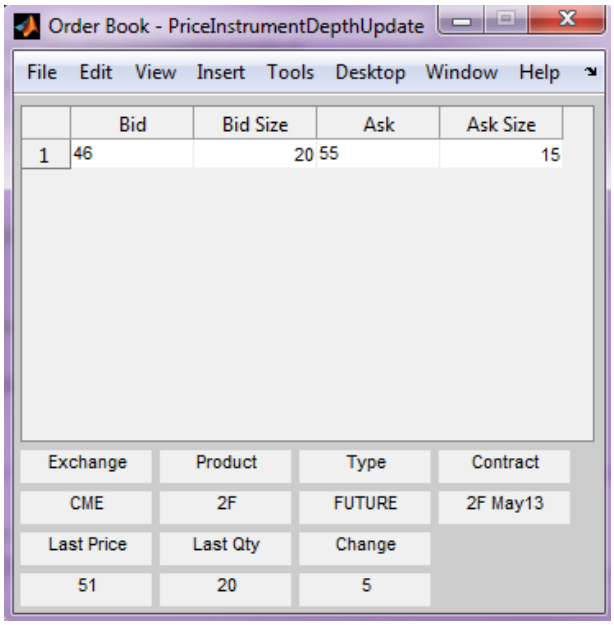
Store Data

```
setappdata(0,'TTOrderBookHandle',uibook)
setappdata(0,'TTOrderBookUIData',ui)
```

Listen for Event Data

Listen for event data with depth updates enabled.

```
X.Instrument(1).Open(1)
```



The last command instructs X_TRADER to start monitoring the attached instruments using the specified event settings.

Close the Connection

```
close(X)
```

See Also

`xtrdr` | `close` | `createInstrument` | `createNotifier` | `getData`

Related Examples

- “Create Order Using X_TRADER” on page 1-13
- “Listen for X_TRADER Price Updates” on page 1-18
- “Submit X_TRADER Orders” on page 1-23

More About

- “Workflows for Trading Technologies X_TRADER” on page 1-16

Submit X_TRADER Orders

This example shows how to connect to X_TRADER and submit orders.

Connect to X_TRADER

```
X = xtrdr;
```

Create an Instrument

```
createInstrument(X,'Exchange','CME','Product','2F',...
                'ProdType','Future','Contract','Aug13',...
                'Alias','SubmitOrderInstrument1')
```

Register Event Handlers

Register event handlers for the order server. The callback `ttorderserverstatus` is assigned to the event `OnExchangeStateUpdate` to verify that the requested instrument's exchange order server is running. Otherwise, no orders can be submitted.

```
sExchange = X.Instrument.Exchange;
registerevent(X.Gate,{'OnExchangeStateUpdate',...
      @(varargin)ttorderserverstatus(varargin{:},sExchange)})
```

Create an Order Set

The `OrderSet` object sends orders to X_TRADER.

Set properties of the `OrderSet` object and detail the level of the order status events. Enable order update and reject (failure) events so you can assign callbacks to handle these conditions.

```
createOrderSet(X)
X.OrderSet(1).EnableOrderRejectData = 1;
X.OrderSet(1).EnableOrderUpdateData = 1;
X.OrderSet(1).OrderStatusNotifyMode = 'ORD_NOTIFY_NORMAL';
```

Set Position Limit Checks

Set whether the order set checks self-imposed position limits when submitting an order.

```
X.OrderSet(1).Set('NetLimits',false)
```

Set a Callback Function

Set a callback to handle the `OnOrderFilled` events. Each time an order is filled (or partially filled), this callback is invoked.

```
registerevent(X.OrderSet(1),{'OnOrderFilled',...
                    @(varargin)ttorderevent(varargin{:},X)})
```

Enable Order Submission

You must first enable order submission before you can submit orders to X_TRADER.

```
X.OrderSet(1).Open(1)
```

Build an Order Profile

Build an order profile using an existing instrument. The order profile contains the settings that define a submitted order. The valid `Set` parameters are shown:

```
orderProfile = createOrderProfile(X);
orderProfile.Instrument = X.Instrument(1);
orderProfile.Customer = '<Default>';
```

Sample: Create a Market Order

Create a market order to buy 100 shares.

```
orderProfile.Set('BuySell','Buy')
orderProfile.Set('Qty',100)
orderProfile.Set('OrderType','M')
```

Sample: Create a Limit Order

Create a limit order by setting the `OrderType` and limit order price.

```
orderProfile.Set('OrderType','L')
orderProfile.Set('Limit$','127000')
```

Sample: Create a Stop Market Order

Create a stop market order and set the order restriction to a stop order and a stop price.

```
orderProfile.Set('OrderType','M')
orderProfile.Set('OrderRestr','S')
orderProfile.Set('Stop$','129800')
```

Sample: Create a Stop Limit Order

Create a stop limit order and set the order restriction, type, limit price, and stop price.

```
orderProfile.Set('OrderType','L')
orderProfile.Set('OrderRestr','S')
orderProfile.Set('Limit$','128000')
orderProfile.Set('Stop$','127500')
```

Check the Order Server Status

Check the order server status before submitting the order and add a counter so the example doesn't delay.

```
nCounter = 1;
while ~exist('bServerUp','var') && nCounter < 20
   pause(1)
   nCounter = nCounter + 1;
end
```

Verify the Order Server Availability

Verify that the exchange's order server in question is available before submitting the order.

```
if exist('bServerUp','var') && bServerUp
   submittedQuantity = X.OrderSet(1).SendOrder(orderProfile);
   disp(['Quantity Sent: ' num2str(submittedQuantity)])
```

```
else
   disp('Order Server is down. Unable to submit order')
end
```

Close the Connection

```
close(X)
```

See Also

`xtrdr` | `close` | `createInstrument` | `createOrderProfile` | `createOrderSet`

Related Examples

- “Create Order Using X_TRADER” on page 1-13
- “Listen for X_TRADER Price Updates” on page 1-18
- “Listen for X_TRADER Price Market Depth Updates” on page 1-20

More About

- “Workflows for Trading Technologies X_TRADER” on page 1-16

Writing and Running Custom Event Handler Functions

Write Custom Event Handler Function

You can process events related to any data updates by writing a custom event handler function for use with Datafeed Toolbox. For example, you can monitor prices before creating an order or plot interval data in a graph. Follow these basic steps to write a custom event handler.

1. Choose the events you want to process, monitor, or evaluate.
2. Decide how the custom event handler processes these events.
3. Determine the input and output arguments for the custom event handler function.
4. Write the code for the custom event handler function.

For details, see “Create Functions in Files”. For a code example of a Bloomberg event handler function, enter `edit v3stockticker.m` at the command line.

Run Custom Event Handler Function

You can run the custom event handler function by passing the function name as an input argument into an existing function. Specify the custom event handler function name either as a character vector, string, or function handle. For details about function handles, see “Create Function Handle”.

For example, suppose you want to retrieve real-time data from Bloomberg using `realtime` with the custom event handler function named `eventhandler`. You can use either of the following syntaxes to run `eventhandler`. This code assumes a Bloomberg connection `c`, security list `s`, Bloomberg data fields `f`, Bloomberg subscription `subs`, and MATLAB timer `t`.

Use a character vector or string.

```
[subs,t] = realtime(c,s,f,'eventhandler');
```

Alternatively, use a function handle.

```
[subs,t] = realtime(c,s,f,@eventhandler);
```

For Bloomberg EMSX interfaces, you can run the custom event handler function by using `timer`. Specify the custom event handler function name as a function handle and pass this function handle as an input argument to `timer`. For details about function handles, see “Create Function Handle”. For example, suppose you want to create an order using `createOrderAndRoute` with the custom event handler function named `eventhandler`. This code assumes a Bloomberg EMSX connection `c`, Bloomberg EMSX order `order`, and timer object `t`.

1. Run `timer` to execute `eventhandler`. The name-value argument `TimerFcn` specifies the event handler function. The name-value argument `Period` specifies a 1-second delay between executions of the event handler function. When the name-value argument `ExecutionMode` is set to `fixedRate`, the event handler function executes immediately after it is added to the MATLAB execution queue.

   ```
   t = timer('TimerFcn',{@c.eventhandler},'Period',1, ...
             'ExecutionMode','fixedRate');
   ```

2. Start the timer to initiate and execute `eventhandler` immediately.

   ```
   start(t)
   ```

3. Run `createOrderAndRoute` using the custom event handler by setting `useDefaultEventHandler` to `false`.

   ```
   createOrderAndRoute(c,order,'useDefaultEventHandler',false)
   ```

4. Stop the timer to stop data updates.

   ```
   stop(t)
   ```

   If you want to resume data updates, run `start`.

5. Delete the timer once you are done with processing data updates for the Bloomberg EMSX connection.

   ```
   delete(t)
   ```

Workflow for Custom Event Handler Function

This workflow summarizes the basic steps to work with a custom event handler function for any of the data service providers, except Bloomberg EMSX.

1. Write a custom event handler function and save it to a file.
2. Create a connection to the data service provider.
3. Subscribe to a specific security using an existing function or API syntax.
4. Run an existing function to receive data updates and use the custom event handler function as an input argument.
5. Stop data updates by using `stop` or by closing the connection to the data service provider.
6. Close the connection to the data service provider if the connection is still open.

For Bloomberg EMSX interfaces, follow this workflow.

1. Write a custom event handler function and save it to a file.
2. Create a connection using `emsx`.
3. Subscribe to Bloomberg EMSX fields using `orders` and `routes`. You can also write custom event handler functions to process subscription events.
4. Run the custom event handler function using `timer`. Use a function handle to specify the custom event handler function name to run `timer`.
5. Start the timer to execute the custom event handler function immediately using `start`.
6. Stop data updates using `stop`.
7. Unsubscribe from Bloomberg EMSX fields by using the API syntax.
8. Delete the timer using `delete`.
9. Close the connection using `close`.

See Also

`realtime` | `close` | `emsx` | `createOrderAndRoute` | `orders` | `routes` | `timer` | `start` | `stop` | `delete`

More About

- “Create Functions in Files”

- “Create Function Handle”

External Websites

- *EMSX API Programmers Guide*

Communicate with Financial Data Servers

- “Communicating with Data Service Providers” on page 2-2
- “Comparing Bloomberg Connections” on page 2-4

Communicating with Data Service Providers

Datafeed Toolbox supports connection to the data providers listed in this table. The table shows the connection functions for each provider.

Provider	Website	Connection Function	API Documentation
Bloomberg	`bloomberg.com`	Bloomberg C++ interfaces: • `bloomberg` • `bloombergServer` • `bloombergBPIPE` • `bloombergEMSX` Bloomberg interfaces: • `blp` • `blpsrv` • `bpipe` • `emsx`	*Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal
CQG	`cqg.com`	`cqg`	CQG API Documentation
FactSet	`factset.com`	`fds`	Not available
FRED	`https://fred.stlouisfed.org`	`fred`	Not available
Haver Analytics	`haver.com`	`haver`	Not available
IHS Markit	`ihsmarkit.com`	`ihsmarkitrs`	IHS Markit Research Signals REST Documentation
Money.Net	`money.net`	`moneynet`	Money.Net API Documentation
Quandl	`data.nasdaq.com`	`quandl`	Quandl Documentation
Refinitiv	`refinitiv.com`	`datastreamws`, `rnseloader`, or `trth`	Not available
SIX Financial Information	`six-group.com`	`tlkrs`	Not available
Trading Technologies	`tradingtechnologies.com`	`xtrdr`	X_TRADER API Documentation
Transaction cost analysis	ftp://ftp.kissellresearch.com	`krg`	Not available
Twitter	`twitter.com`	`twitter`	Twitter REST API Endpoint Reference Documentation

Provider	Website	Connection Function	API Documentation
Wind Data Feed Services (WDS)	https://www.wind.com.cn/NewSite/data.html	`wind`	Not available

See Also

More About

- “Data Server Connection Requirements” on page 1-3

Comparing Bloomberg Connections

Datafeed Toolbox uses these different Bloomberg services to connect to Bloomberg interfaces and Bloomberg C++ interfaces. To learn about the connection functions and the data access functionality of each service, see this table.

You need a valid Bloomberg license to work with each Bloomberg service.

Bloomberg Service	Bloomberg Desktop	Bloomberg Server	Bloomberg B-PIPE
Connection function	`blp` (Bloomberg interface) or `bloomberg` (Bloomberg C++ interface)	`blpsrv` (Bloomberg interface) or `bloombergServer` (Bloomberg C++ interface)	`bpipe` (Bloomberg interface) or `bloombergBPIPE` (Bloomberg C++ interface)
Data access	Applications obtain data from the Bloomberg Data Center by connecting locally to the Bloomberg Communications Server	Applications obtain data from the Bloomberg Data Center using a dedicated process that optimizes network resources	Provides entitled users access to permission data from the Bloomberg Data Center through the Bloomberg Appliance

To manage Bloomberg orders and obtain broker information, you can connect to Bloomberg EMSX using the `emsx` (Bloomberg interface) or `bloombergEMSX` (Bloomberg C++ interface) functions.

Each connection function has a different syntax for creating a Bloomberg connection. The connection objects created by running these functions have different properties. For details, see the respective function reference page.

For details about these services, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

See Also

Related Examples

- "Connect to Bloomberg" on page 3-2

More About

- "Installing Bloomberg and Configuring Connections" on page 1-5
- "Data Server Connection Requirements" on page 1-3

Data Provider Workflows

Connect to Bloomberg

This example shows how to create a connection to Bloomberg using these Bloomberg services: Bloomberg Desktop, Bloomberg Server, and B-PIPE. For details about Bloomberg connection requirements, see “Data Server Connection Requirements” on page 1-3. To ensure a successful Bloomberg connection, perform the required steps before executing a connection function. For details, see “Installing Bloomberg and Configuring Connections” on page 1-5.

Create the Bloomberg Desktop Connection

```
c = blp
```

```
c =
  blp with properties:
      session: [1x1 com.bloomberglp.blpapi.Session]
    ipaddress: 'localhost'
         port: 8194
      timeout: 0
```

`blp` creates a Bloomberg connection object `c` and returns its properties.

Validate the connection `c`.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Retrieve the Bloomberg Desktop connection properties.

```
v = get(c)
```

```
v =

      session: [1x1 com.bloomberglp.blpapi.Session]
    ipaddress: 'localhost'
         port: 8194
      timeout: 0
```

`v` is a structure containing the Bloomberg session object, IP address, port number, and timeout value.

Close the Bloomberg Desktop connection.

```
close(c)
```

Create the Bloomberg Server Connection

Connect to the Bloomberg Server using the IP addresses of the machine running the Bloomberg Server. This code assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address `serverip` for the machine running the Bloomberg Server is `'111.11.11.111'`.

```
uuid = 12345678;
serverip = '111.11.11.111';
```

```
c = blpsrv(uuid,serverip)

c =

  blpsrv with properties:

         uuid: 12345678
         user: [1x1 com.bloomberglp.blpapi.impl.aT]
      session: [1x1 com.bloomberglp.blpapi.Session]
    ipaddress: '111.11.11.111'
         port: 8195
      timeout: 0
```

`blpsrv` connects to the machine running the Bloomberg Server on the default port number `8195`. `blpsrv` creates the Bloomberg Server connection object `c`.

Close the Bloomberg Server connection.

```
close(c)
```

Create the B-PIPE Connection

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This code assumes the following:

- The authentication is Windows Authentication by setting `authorizationtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address `serverip` for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number is `8194`.

```
authorizationtype = 'OS_LOGON';
applicationname = '';
serverip = {'111.11.11.112'};
portnumber = 8194;

c = bpipe(authorizationtype,applicationname,serverip,portnumber)

c =

  bpipe with properties:

    appauthtype: ''
       authtype: 'OS_LOGON'
        appname: []
           user: [1x1 com.bloomberglp.blpapi.impl.aT]
        session: [1x1 com.bloomberglp.blpapi.Session]
      ipaddress: {'111.11.11.112'}
           port: 8194.00
        timeout: 0
```

`bpipe` connects to Bloomberg B-PIPE at the port number `8194`. `bpipe` creates the Bloomberg B-PIPE connection object `c`.

Close the B-PIPE connection.

```
close(c)
```

See Also

`blp` | `blpsrv` | `bpipe` | `get` | `isconnection` | `close`

Related Examples

- “Retrieve Bloomberg Current Data” on page 3-5
- “Retrieve Bloomberg Historical Data” on page 3-7
- “Retrieve Current and Historical Data Using Bloomberg” on page 1-7
- “Retrieve Bloomberg Intraday Tick Data” on page 3-11
- “Retrieve Bloomberg Real-Time Data” on page 3-13

More About

- “Data Server Connection Requirements” on page 1-3
- “Comparing Bloomberg Connections” on page 2-4
- “Workflow for Bloomberg” on page 3-15

Retrieve Bloomberg Current Data

This example shows how to retrieve current data from Bloomberg for a single security and for multiple securities. To create a successful Bloomberg connection, see “Connect to Bloomberg” on page 3-2.

Connect to Bloomberg

Create a Bloomberg Desktop connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve Current Data for Single Security

Retrieve last and open prices for Microsoft®.

`d` contains the Bloomberg last and open prices as fields in a structure. `sec` contains the Bloomberg security name for Microsoft in a cell array. The security name is a character vector.

```
sec = 'MSFT US Equity';
fields = {'LAST_PRICE';'OPEN'}; % Retrieve data for last and open prices

[d,sec] = getdata(c,sec,fields)
```

```
d =

  struct with fields:

    LAST_PRICE: 62.30
          OPEN: 62.95


sec =

  cell

    'MSFT US Equity'
```

Retrieve Current Data for Multiple Securities

Retrieve last and open prices for the IBM® and Ford Motor Company® securities.

`d` contains the Bloomberg last and open prices as fields in a structure. `sec` contains the Bloomberg security names for IBM and Ford Motor Company in a cell array. Each security name is a character vector.

```
s = {'IBM US Equity','F US Equity'};
fields = {'LAST_PRICE';'OPEN'}; % Retrieve data for last and open prices

[d,sec] = getdata(c,s,fields)
```

```
d =

  struct with fields:
```

```
    LAST_PRICE: [2×1 double]
          OPEN: [2×1 double]


sec =

  2×1 cell array

    'IBM US Equity'
    'F US Equity'
```

Display the last price for both securities.

```
d.LAST_PRICE

ans =

        166.73
         12.63
```

Close Bloomberg Connection

```
close(c)
```

See Also

`blp` | `getdata` | `close`

Related Examples

- “Connect to Bloomberg” on page 3-2
- “Retrieve Bloomberg Historical Data” on page 3-7
- “Retrieve Current and Historical Data Using Bloomberg” on page 1-7
- “Retrieve Bloomberg Intraday Tick Data” on page 3-11
- “Retrieve Bloomberg Real-Time Data” on page 3-13

More About

- “Workflow for Bloomberg” on page 3-15

Retrieve Bloomberg Historical Data

This example shows how to retrieve historical data from Bloomberg for a single security. The example shows retrieving weekly data within a date range and retrieving data with a default period. Then, the example also shows how to retrieve data for multiple securities. To create a successful Bloomberg connection, see “Connect to Bloomberg” on page 3-2.

Connect to Bloomberg

Create a Bloomberg Desktop connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve Historical Data for One Security

Retrieve monthly closing and open price data from January 1, 2012, through December 31, 2012, for Microsoft.

```
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
fromdate = '1/01/2012';
todate = '12/31/2012';
period = 'monthly';

[d,sec] = history(c,s,f,fromdate,todate,period)

d =

     734899.00         27.87         25.06
     734928.00         30.16         28.12
     734959.00         30.65         30.34
     ...

sec =

  cell

    'MSFT US Equity'
```

`d` contains the numeric representation of the date in the first column, closing price in the second column, and open price in the third column. Each row represents data for one month in the date range. `sec` contains the Bloomberg security name for Microsoft.

Retrieve Weekly Historical Data

Retrieve the weekly closing prices from November 1, 2010, through December 23, 2010, for the Microsoft security using US currency. In this case, the anchor date depends on the date December 23, 2010. Because this date is a Thursday, each previous value is reported for the Thursday of the week in question.

```
f = 'LAST_PRICE';
fromdate = '11/01/2010';
todate =  '12/23/2010';
period = {'weekly'};
currency = 'USD';
```

```
[d,sec] = history(c,s,f,fromdate,todate, ...
    period,currency)

d =

     734446.00         27.14
     734453.00         26.68
     734460.00         25.84
     734467.00         25.37
     734474.00         26.89
     734481.00         27.08
     734488.00         27.99
     734495.00         28.30


sec =

  1×1 cell array

    {'MSFT US Equity'}
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the Microsoft security.

Retrieve Historical Data Using Default Period

Retrieve the closing prices from August 1, 2010, through September 10, 2010, for the Microsoft security in US currency, and set the default period of the data by using `[]`. The default period of a security depends on the security itself.

```
fromdate = '8/01/2010';
todate = '9/10/2010';
currency = 'USD';

[d,sec] = history(c,s,f,fromdate,todate, ...
    [],currency)

d =

     734352.00         26.33
     734353.00         26.16
     734354.00         25.73
     ...

sec =

  1×1 cell array

    {'MSFT US Equity'}
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the Microsoft security.

Retrieve Historical Data for Multiple Securities

Retrieve monthly closing and open prices from January 1, 2012, through December 31, 2012, for the IBM and Ford Motor Company securities.

d is a cell array of double matrices that contains the historical data for both securities. sec contains the Bloomberg security names for the IBM and Ford Motor Company securities in a cell array. Each security name is a character vector.

```
s = {'IBM US Equity','F US Equity'};
f = {'LAST_PRICE';'OPEN'};
fromdate = '1/01/2012';
todate = '12/31/2012';
period = 'monthly';

[d,sec] = history(c,s,f,fromdate,todate,period)

d =

  2×1 cell array

    [12×3 double]
    [12×3 double]


sec =

  2×1 cell array

    'IBM US Equity'
    'F US Equity'
```

Display the closing and open prices for the first security.

```
d{1}

ans =

     734899.00        192.60        186.73
     734928.00        196.73        193.21
     734959.00        208.65        197.23
     ...
```

The data in the double matrix is:

- First column — Numeric representation of the date
- Second column — Closing price
- Third column — Open price

Each row represents data for one month in the date range.

Close Bloomberg Connection

```
close(c)
```

See Also

blp | history | close

Related Examples

- “Connect to Bloomberg” on page 3-2
- “Retrieve Bloomberg Current Data” on page 3-5

- “Retrieve Current and Historical Data Using Bloomberg” on page 1-7
- “Retrieve Bloomberg Intraday Tick Data” on page 3-11
- “Retrieve Bloomberg Real-Time Data” on page 3-13

More About

- “Workflow for Bloomberg” on page 3-15

Retrieve Bloomberg Intraday Tick Data

This example shows how to retrieve intraday tick data from Bloomberg. To create a successful Bloomberg connection, see “Connect to Bloomberg” on page 3-2.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the trade tick series for the past 50 days for the IBM security aggregated into 5-minute intervals.

```
d = timeseries(c,'IBM US Equity',{floor(now)-50,floor(now)},5,'Trade')
```

```
ans =

  Columns 1 through 7

   735487.40        187.20        187.60        187.02        187.08     207683.00        560.00
   735487.40        187.03        187.13        186.65        186.78      46990.00        349.00
   735487.40        186.78        186.78        186.40        186.47      51589.00        399.00
     ...

Column 8

   38902968.00
    8779374.00
    9626896.00
   ...
```

The columns in `d` contain the following:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row of data shows prices and tick data for the current date. The next row shows tick data for 5 minutes later.

Close the Bloomberg connection.

```
close(c)
```

See Also

`blp` | `timeseries` | `close`

Related Examples

- “Connect to Bloomberg” on page 3-2
- “Retrieve Bloomberg Current Data” on page 3-5
- “Retrieve Bloomberg Historical Data” on page 3-7
- “Retrieve Current and Historical Data Using Bloomberg” on page 1-7
- “Retrieve Bloomberg Real-Time Data” on page 3-13

More About

- “Workflow for Bloomberg” on page 3-15

Retrieve Bloomberg Real-Time Data

This example shows how to retrieve real-time data from Bloomberg. To create a successful Bloomberg connection, see “Connect to Bloomberg” on page 3-2. Here, to return Bloomberg stock tick data, use the event handler `v3stockticker`. Instead of the default event handler, you can create your own event handler function to process Bloomberg data.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the last trade and volume for IBM and Ford Motor Company securities.

`v3stockticker` requires the input argument `f` of the `realtime` function to be `'Last_Trade'`, `'Volume'`, or both.

```
[subs,t] = realtime(c,{'IBM US Equity','F US Equity'}, ...
                    {'Last_Trade','Volume'},'v3stockticker')
```

```
subs =

com.bloomberglp.blpapi.SubscriptionList@6c1066f6


   Timer Object: timer-3

   Timer Settings
      ExecutionMode: fixedRate
             Period: 0.05
           BusyMode: drop
            Running: on

   Callbacks
           TimerFcn: 1x4 cell array
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''

** IBM US Equity ** 32433 @ 181.85 29-Oct-2013 15:50:05
** IBM US Equity ** 200 @ 181.85 29-Oct-2013 15:50:05
** IBM US Equity ** 100 @ 181.86 29-Oct-2013 15:50:05
** F US Equity ** 300 @ 17.575 30-Oct-2013 10:14:06
** F US Equity ** 100 @ 17.57 30-Oct-2013 10:14:06
** F US Equity ** 100 @ 17.5725 30-Oct-2013 10:14:06
...
```

`realtime` returns the Bloomberg subscription list object `subs` and the MATLAB timer object with its properties. Then, `realtime` returns the stock tick data for the IBM and Ford Motor Company securities with the last trade price and volume.

Real-time data continues to display until you use the `stop` or `close` function.

Close the Bloomberg connection.

```
close(c)
```

See Also

`blp` | `close` | `realtime` | `stop`

Related Examples

- “Connect to Bloomberg” on page 3-2
- “Retrieve Bloomberg Current Data” on page 3-5
- “Retrieve Bloomberg Historical Data” on page 3-7
- “Retrieve Current and Historical Data Using Bloomberg” on page 1-7
- “Retrieve Bloomberg Intraday Tick Data” on page 3-11

More About

- “Workflow for Bloomberg” on page 3-15
- “Writing and Running Custom Event Handler Functions” on page 1-26

Workflow for Bloomberg

You can use Bloomberg to monitor market price information.

Bloomberg Desktop, Bloomberg Server, or Bloomberg B-PIPE Services

Connect to Bloomberg

1. Connect to Bloomberg using `blp`, `blpsrv`, or `bpipe`.
2. Ensure a successful Bloomberg connection by using `isconnection`. Request properties of the connection objects using `get`.

Request Current, Historical, Intraday Tick, Portfolio, or Real-Time Data

1. Look up information about securities, curves, or government securities using `lookup`. Request Bloomberg field information using `category`, `fieldinfo`, or `fieldsearch`.
2. Request current data for a security using `getdata`. Request bulk data with header information using `getbulkdata`.
3. Request equity screening data using `eqs`.
4. Request historical data for a security using `history`.
5. Request historical technical analysis using `tahistory`.
6. Request intraday tick data for a security using `timeseries`.
7. Request current portfolio data using `portfolio`.
8. Request real-time data for a security using `realtime`. Stop real-time data updates using `stop`.

Close Bloomberg Connection

Close the Bloomberg connection by using `close`.

See Also

Related Examples

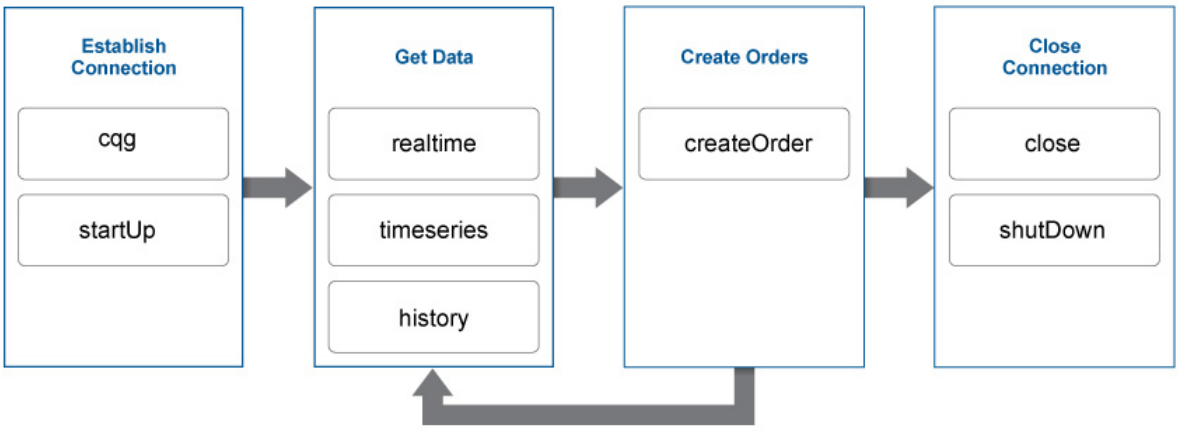
- “Connect to Bloomberg” on page 3-2
- “Retrieve Bloomberg Current Data” on page 3-5
- “Retrieve Bloomberg Historical Data” on page 3-7
- “Retrieve Bloomberg Intraday Tick Data” on page 3-11
- “Retrieve Bloomberg Real-Time Data” on page 3-13

More About

- “Comparing Bloomberg Connections” on page 2-4

Workflow for CQG

This diagram shows the functions you can use with CQG to monitor market price information and submit orders.



To request current, intraday, or historical data:

1. Create the CQG connection object using `cqg`.
2. Define the CQG event handlers.
3. Connect to CQG using `startUp`.
4. Subscribe to a CQG instrument to request real-time data using `realtime`.
5. Request intraday data for a security using `timeseries`.
6. Request historical data for a security using `history`.
7. Close the CQG connection using `close` or `shutDown`.

To submit orders to CQG:

1. Create the CQG connection object using `cqg`.
2. Define the CQG event handlers.
3. Connect to CQG using `startUp`.
4. Create the CQG account credentials object.
5. Subscribe to a CQG instrument to request real-time data using `realtime`.
6. Create and submit the order using `createOrder`.
7. Close the CQG connection using `close` or `shutDown`.

See Also

Related Examples

- “Create Order Using CQG” on page 3-18
- “Create CQG Orders” on page 3-20

- “Request CQG Historical Data” on page 3-24
- “Request CQG Intraday Tick Data” on page 3-27
- “Request CQG Real-Time Data” on page 3-30

Create Order Using CQG

This example shows how to connect to CQG and create a market order.

Connect to CQG

```
c = cqg;
```

Establish Event Handlers

Start the CQG session. Set up event handlers for instrument subscription, orders, and associated events.

```
startUp(c)

streamEventNames = {'InstrumentSubscribed', ...
    'InstrumentChanged','IncorrectSymbol'};

for i = 1:length(streamEventNames)
  registerevent(c.Handle,{streamEventNames{i}, ...
    @(varargin)cqgrealtimeeventhandler(varargin{:})})
end

orderEventNames = {'AccountChanged','OrderChanged','AllOrdersCanceled'};

for i = 1:length(orderEventNames)
  registerevent(c.Handle,{orderEventNames{i}, ...
    @(varargin)cqgordereventhandler(varargin{:})})
end
```

Subscribe to Instrument

Subscribe to a security tied to the EURIBOR.

```
realtime(c,'F.US.IE')
pause(2)
```

Create `CQGInstrument` Object

To use the instrument for creating an order, import the instrument name `cqgInstrumentName` into the current MATLAB workspace. Then, create the `CQGInstrument` object `cqgInst`.

```
cqgInstrumentName = evalin('base','cqgInstrument');
cqgInst = c.Handle.Instruments.Item(cqgInstrumentName);
```

Set Up Account Credentials

Set the CQG flags to enable account information retrieval.

```
c.Handle.set('AccountSubscriptionLevel','aslNone');
c.Handle.set('AccountSubscriptionLevel','aslAccountUpdatesAndOrders');
pause(2)
accountHandle = c.Handle.Accounts.ItemByIndex(0);
```

Create Market Order

Create a market order that buys one share of the subscribed security `cqgInst` using the account credentials `accountHandle`.

```
orderType = 1;  % Market order flag
quantity = 1;   % Positive quantity is Buy, negative is Sell
oMarket = createOrder(c,cqgInst,orderType,accountHandle,quantity);
oMarket.Place
```

Close CQG Connection

```
close(c)
```

See Also

`cqg` | `close` | `createOrder` | `realtime` | `startUp`

Related Examples

- “Create CQG Orders” on page 3-20
- “Request CQG Historical Data” on page 3-24
- “Request CQG Intraday Tick Data” on page 3-27
- “Request CQG Real-Time Data” on page 3-30

More About

- “Workflow for CQG” on page 3-16

External Websites

- *CQG API Reference Guide*

Create CQG Orders

This example shows how to connect to CQG, define the event handlers, subscribe to the security, define the account handle, and submit orders for execution.

Create the CQG Connection

Create the CQG connection object using `cqg`.

```
c = cqg;
```

Define Event Handlers

Register the sample event handler `cqgconnectioneventhandler` to track events associated with the connection status.

```
eventNames = {'CELStarted','DataError','IsReady', ...
    'DataConnectionStatusChanged', ...
    'GWConnectionStatusChanged', ...
    'GWEnvironmentChanged'};

for i = 1:length(eventNames)
  registerevent(c.Handle,{eventNames{i}, ...
    @(varargin)cqgconnectioneventhandler(varargin{:})})
end
```

`cqgconnectioneventhandler` is assigned to the events in `eventNames`.

Set the API configuration properties. For example, to set the time zone to Eastern Time, enter the following.

```
c.APIConfig.TimeZoneCode = 'tzEastern';
```

`c.APIConfig` is a CQG configuration object. For details about setting API configuration properties, see *CQG API Reference Guide*.

Establish the connection to CQG.

```
startUp(c)
```

```
CELStarted
DataConnectionStatusChanged
GWConnectionStatusChanged
```

The connection event handler displays event names for a successful CQG connection.

Register an event handler to track events associated with a CQG instrument subscription.

```
streamEventNames = {'InstrumentSubscribed','InstrumentChanged', ...
    'IncorrectSymbol'};

for i = 1:length(streamEventNames)
  registerevent(c.Handle,{streamEventNames{i}, ...
    @(varargin)cqgrealtimeeventhandler(varargin{:})})
end
```

Register an event handler to track events associated with a CQG order and account.

```
orderEventNames = {'AccountChanged','OrderChanged','AllOrdersCanceled'};

for i = 1:length(orderEventNames)
  registerevent(c.Handle,{orderEventNames{i}, ...
    @(varargin)cqgordereventhandler(varargin{:})})
end
```

Subscribe to the CQG Instrument

With the connection established, subscribe to the CQG instrument. The instrument must be successfully subscribed first before it is available for transactions. You must format the instrument name in the CQG long symbol view. For example, to subscribe to a security tied to the EURIBOR, enter the following.

```
realtime(c,'F.US.IE')
pause(2)
```

```
F.US.IEK13 subscribed
```

`pause` causes MATLAB to wait 2 seconds before continuing to give time for CQG to subscribe to the instrument.

Create the CQG instrument object.

To use the instrument in `createOrder`, import the name of the instrument `cqgInstrumentName` into the current MATLAB workspace. Then, create the `CQGInstrument` object `cqgInst`.

```
cqgInstrumentName = evalin('base','cqgInstrument');
cqgInst = c.Handle.Instruments.Item(cqgInstrumentName);
```

Set Up Account Credentials

Set the CQG flags to enable account information retrieval.

```
set(c.Handle,'AccountSubscriptionLevel','aslNone')
set(c.Handle,'AccountSubscriptionLevel','aslAccountUpdatesAndOrders')
pause(2)
```

```
ans =
    AccountChanged
```

The CQG API shows that account information changed.

Set up the CQG account credentials.

Retrieve the `CQGAccount` object into `accountHandle` to use your account information in `createOrder`. For details about creating a `CQGAccount` object, see *CQG API Reference Guide*.

```
accountHandle = c.Handle.Accounts.ItemByIndex(0);
```

Create CQG Market, Limit, Stop, and Stop Limit Orders

Create a market order that buys one share of the subscribed security `cqgInst` using the account credentials `accountHandle`.

```
quantity = 1;

oMarket = createOrder(c,cqgInst,1,accountHandle,quantity);
oMarket.Place
```

```
ans =
    OrderChanged
```

The `CQGOrder` object `oMarket` contains the order. The CQG API executes the market order using the CQG API function `Place`. After execution, the order status changes.

To use a character vector for the security, subscribe to the security `'EZC'` as shown above. Then, create a market order that buys one share of the security `'EZC'` using the defined account credentials `accountHandle`.

```
cqgInstrumentName = 'EZC';
quantity = 1;

oMarket = createOrder(c,cqgInstrumentName,1,accountHandle,quantity);
oMarket.Place
```

```
ans =
    OrderChanged
```

The `CQGOrder` object `oMarket` contains the order. The CQG API executes the market order using the CQG API function `Place`. After execution, the order status changes.

To create a limit order, you can use the bid price. Extract the CQG bid object `qtBid` from the previously defined `CQGInstrument` object `cqgInst`. For details about the `CQGInstrument` object, see *CQG API Reference Guide*.

```
qtBid = cqgInst.get('Bid');
```

Create a limit order that buys one share of the previously subscribed security `cqgInst` using the previously defined account credentials `accountHandle` and `qtBid` for the limit price.

```
quantity = 1;
limitprice = qtBid.get('Price');

oLimit = createOrder(c,cqgInst,2,accountHandle,quantity,limitprice);
oLimit.Place
```

```
ans =
    OrderChanged
```

The `CQGOrder` object `oLimit` contains the order. The CQG API executes the limit order using the CQG API function `Place`. After execution, the order status changes.

To create a stop order, you can use the trade price. Extract the CQG trade object `qtTrade` from the previously defined `CQGInstrument` object `cqgInst`.

```
qtTrade = cqgInst.get('Trade');
```

Create a stop order that buys one share of the previously subscribed security `cqgInst` using the previously defined account credentials `accountHandle` and `qtTrade` for the stop price.

```
quantity = 1;
stopprice = qtTrade.get('Price');

oStop = createOrder(c,cqgInst,3,accountHandle,quantity,stopprice);
oStop.Place
```

```
ans =
    OrderChanged
```

The `CQGOrder` object `oStop` contains the order. The CQG API executes the stop order using the CQG API function `Place`. After execution, the order status changes.

To create a stop limit order, use both the bid and trade prices defined above. Create a stop limit order that buys one share of the subscribed security `cqgInst` using the defined account credentials `accountHandle`.

```
quantity = 1;

oStopLimit = createOrder(c,cqgInst,4,accountHandle,quantity, ...
    limitprice,stopprice);
oStopLimit.Place
```

```
ans =
    OrderChanged
```

The `CQGOrder` object `oStopLimit` contains the order. The CQG API executes the stop limit order using the CQG API function `Place`. After execution, the order status changes.

Close the CQG Connection

```
shutDown(c)
```

See Also

`cqg` | `close` | `createOrder` | `history` | `timeseries` | `startUp` | `shutDown` | `realtime`

Related Examples

- “Create Order Using CQG” on page 3-18
- “Request CQG Historical Data” on page 3-24
- “Request CQG Real-Time Data” on page 3-30
- “Request CQG Intraday Tick Data” on page 3-27

More About

- “Workflow for CQG” on page 3-16

External Websites

- *CQG API Reference Guide*

Request CQG Historical Data

This example shows how to connect to CQG, define event handlers, and request historical data.

Connect to CQG

Create the CQG connection object using `cqg`.

```
c = cqg;
```

Define Event Handlers

Register the sample event handler `cqgconnectioneventhandler` to track events associated with connection status.

```
eventNames = {'CELStarted','DataError','IsReady', ...
    'DataConnectionStatusChanged'};

for i = 1:length(eventNames)
  registerevent(c.Handle,{eventNames{i}, ...
    @(varargin)cqgconnectioneventhandler(varargin{:})})
end
```

`cqgconnectioneventhandler` is assigned to the events in `eventNames`.

Set the API configuration properties. For example, to set the time zone to Eastern Time, enter the following.

```
c.APIConfig.TimeZoneCode = 'tzEastern';
```

`c.APIConfig` is a CQG configuration object. For details about setting API configuration properties, see *CQG API Reference Guide*.

Create the CQG connection.

```
startUp(c)
```

```
CELStarted
DataConnectionStatusChanged
```

The connection event handler displays event names for a successful CQG connection.

Register an event handler to build and initialize the output data matrix `cqgHistoryData`.

```
histEventNames = {'ExpressionResolved','ExpressionAdded', ...
    'ExpressionUpdated'};

for i = 1:length(histEventNames)
  registerevent(c.Handle,{histEventNames{i}, ...
    @(varargin)cqgexpressioneventhandler(varargin{:})})
end
```

Pass an Additional Optional Request Property

Pass an additional optional request property by creating the structure `x` and setting the optional property.

```
x.UpdatesEnabled = false;
```

For additional optional properties you can set, see *CQG API Reference Guide*.

Request CQG Historical Data

Request daily data for instrument `XYZ.XYZ` for the last 10 days using the additional optional request property `x`. `XYZ.XYZ` is a sample instrument name. To request historical data for your instrument, substitute the symbol name in `instrument`.

```
instrument = {'Close(XYZ.XYZ)','Open(XYZ.XYZ)'};
startdate = floor(now) - 10;
enddate = floor(now);
period = 'hpDaily';

history(c,instrument,startdate,enddate,period,x)
pause(1)
```

MATLAB writes the variable `cqgHistoryData` to the Workspace browser.

Display `cqgHistoryData`.

```
cqgHistoryData
```

```
cqgHistoryData =
   1.0e+05 *
    7.3533    0.0063    0.0063
    7.3533    0.0064    0.0064
    7.3533    0.0065    0.0065
    7.3534    0.0065    0.0065
    7.3534    0.0066    0.0066
    7.3534    0.0065    0.0065
    7.3534    0.0066    0.0066
    7.3534    0.0066    0.0066
    7.3534    0.0064    0.0064
```

Each row in `cqgHistoryData` represents data for 1 day. The columns in `cqgHistoryData` show the numerical representation of the timestamp, the close price, and the open price for the instrument during the day.

Close the CQG Connection

```
close(c)
```

See Also

`cqg` | `close` | `createOrder` | `history` | `timeseries` | `startUp` | `shutDown` | `realtime`

Related Examples

- “Create Order Using CQG” on page 3-18
- “Create CQG Orders” on page 3-20
- “Request CQG Real-Time Data” on page 3-30
- “Request CQG Intraday Tick Data” on page 3-27

More About

- “Workflow for CQG” on page 3-16

External Websites

- *CQG API Reference Guide*

Request CQG Intraday Tick Data

This example shows how to connect to CQG, define event handlers, and request intraday and timed bar data.

Connect to CQG and Define Event Handlers

Create the CQG connection object using `cqg`.

```
c = cqg;
```

Register the sample event handler `cqgconnectioneventhandler` to track events associated with the connection status.

```
eventNames = {'CELStarted','DataError','IsReady', ...
    'DataConnectionStatusChanged'};

for i = 1:length(eventNames)
  registerevent(c.Handle,{eventNames{i}, ...
    @(varargin)cqgconnectioneventhandler(varargin{:})})
end
```

`cqgconnectioneventhandler` is assigned to the events in `eventNames`.

Set the API configuration properties. For example, to set the time zone to Eastern Time, enter the following.

```
c.APIConfig.TimeZoneCode = 'tzEastern';
```

`c.APIConfig` is a CQG configuration object. For details about setting API configuration properties, see *CQG API Reference Guide*.

Create the CQG connection.

```
startUp(c)
```

```
CELStarted
DataConnectionStatusChanged
```

The connection event handler displays event names for a successful CQG connection.

Register an event handler to build and initialize the output data structure `cqgTickData` used for storing intraday tick data.

```
rawEventNames = {'TicksResolved','TicksAdded'};

for i = 1:length(rawEventNames)
  registerevent(c.Handle,{rawEventNames{i}, ...
    @(varargin)cqgintradayeventhandler(varargin{:})})
end
```

Request CQG Intraday Tick Data

Pass an additional optional request property by creating the structure `x`, and setting the optional property. To see only bid tick data, for example, set `TickFilter` to `'tfBid'`.

```
x.TickFilter = 'tfBid';
```

`TickFilter` and `SessionsFilter` are the only valid additional optional properties for calling `timeseries` without a timed bar request. For additional property values you can set, see *CQG API Reference Guide*.

Request intraday tick data for instrument `XYZ.XYZ` for the last 2 days using the additional optional request property `x`. `XYZ.XYZ` is a sample instrument name. To request intraday tick data for your instrument, substitute the symbol name in `instrument`.

```
instrument = 'XYZ.XYZ';
startdate = now - 2;
enddate = now;

timeseries(c,instrument,startdate,enddate,[],x)
pause(1)
```

`pause` causes MATLAB to wait 1 second before continuing to give time for CQG to subscribe to the instrument. MATLAB writes the variable `cqgTickData` to the Workspace browser.

Display `cqgTickData`.

```
cqgTickData
```

```
cqgTickData =
              Timestamp: {2x1 cell}
                  Price: [2x1 double]
                 Volume: [2x1 double]
              PriceType: {2x1 cell}
         CorrectionType: {2x1 cell}
    SalesConditionLabel: {2x1 cell}
     SalesConditionCode: [2x1 double]
          ContributorId: {2x1 cell}
      ContributorIdCode: [2x1 double]
            MarketState: {2x1 cell}
```

Display data in the `Timestamp` property of `cqgTickData`.

```
cqgTickData.Timestamp
```

```
ans =
    '4/17/2013 2:14:00 PM'
    '4/18/2013 2:14:00 PM'
```

Request CQG Timed Bar Data

Register an event handler to build and initialize the output data matrix `cqgTimedBarData` used for storing timed bar data.

```
aggEventNames = {'TimedBarsResolved','TimedBarsAdded', ...
    'TimedBarsUpdated','TimedBarsInserted', ...
    'TimedBarsRemoved'};

for i = 1:length(aggEventNames)
  registerevent(c.Handle,{aggEventNames{i}, ...
    @(varargin)cqgintradayeventhandler(varargin{:})})
end
```

Pass additional optional request properties by creating the structure `x`, and setting the optional property.

```
x.UpdatesEnabled = false;
```

Request timed bar data for instrument `XYZ.XYZ` for the last fraction of a day using the additional optional request property `x`. `XYZ.XYZ` is a sample instrument name. To request timed bar data for your instrument, substitute the symbol name in `instrument`.

```
instrument = 'XYZ.XYZ';
startdate = now - .1;
enddate = now;
intraday = 1;

timeseries(c,instrument,startdate,enddate,intraday,x)
pause(1)
```

MATLAB writes the variable `cqgTimedBarData` to the Workspace browser.

Display `cqgTimedBarData`.

```
cqgTimedBarData
```

```
cqgTimedBarData =
   1.0e+09 *
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    ...
```

`cqgTimedBarData` returns timed bar data for the specified instrument. The columns of `cqgTimedBarData` display data corresponding to the timestamp, open price, high price, low price, close price, mid-price, HLC3, average price, and tick volume.

Close the CQG Connection

```
close(c)
```

See Also

`cqg` | `close` | `createOrder` | `history` | `timeseries` | `startUp` | `shutDown` | `realtime`

Related Examples

- “Create Order Using CQG” on page 3-18
- “Create CQG Orders” on page 3-20
- “Request CQG Historical Data” on page 3-24
- “Request CQG Real-Time Data” on page 3-30

More About

- “Workflow for CQG” on page 3-16

External Websites

- *CQG API Reference Guide*

Request CQG Real-Time Data

This example shows how to connect to CQG, define event handlers, and request current data.

Connect to CQG

Create the CQG connection object using `cqg`.

```
c = cqg;
```

Define Event Handlers

Register the sample event handler `cqgconnectioneventhandler` to track events for the connection status.

```
eventNames = {'CELStarted','DataError','IsReady', ...
    'DataConnectionStatusChanged','GWConnectionStatusChanged', ...
    'GWEnvironmentChanged'};

for i = 1:length(eventNames)
  registerevent(c.Handle,{eventNames{i}, ...
    @(varargin)cqgconnectioneventhandler(varargin{:})})
end
```

`cqgconnectioneventhandler` is assigned to the events in `eventNames`.

Set the API configuration properties. For example, to set the time zone to Eastern Time, enter the following.

```
c.APIConfig.TimeZoneCode = 'tzEastern';
```

`c.APIConfig` is a CQG configuration object. For details about setting the API configuration properties, see *CQG API Reference Guide*.

Establish the connection to CQG.

```
startUp(c)
```

```
CELStarted
DataConnectionStatusChanged
GWConnectionStatusChanged
```

The connection event handler displays event names for a successful CQG connection.

Register an event handler to track events associated with the CQG instrument subscription.

```
streamEventNames = {'InstrumentSubscribed','InstrumentChanged', ...
    'IncorrectSymbol'};

for i = 1:length(streamEventNames)
  registerevent(c.Handle,{streamEventNames{i}, ...
    @(varargin)cqgrealtimeeventhandler(varargin{:})})
end
```

Request CQG Real-Time Data

With the connection established, subscribe to the instrument. The instrument name must be formatted in the CQG long symbol view. For example, to subscribe to a security tied to corn, enter the

following. (`F.US.EZC` is a sample instrument name. To request real-time data for your instrument, substitute this sample name with the name of your instrument.)

```
instrument = 'F.US.EZC';
realtime(c,instrument)
```

MATLAB writes the structure variable `cqgDataEZC` to the Workspace browser.

Display `cqgDataEZC`.

```
cqgDataEZC(1,1)
```

```
ans =
              Price: {15x1 cell}
             Volume: {15x1 cell}
    ServerTimestamp: {15x1 cell}
          Timestamp: {15x1 cell}
               Type: {15x1 cell}
               Name: {15x1 cell}
            IsValid: {15x1 cell}
         Instrument: {15x1 cell}
          HasVolume: {15x1 cell}
```

`cqgDataEZC` returns the current quotes for the security.

Display data in the `Price` property of `cqgDataEZC`.

```
cqgDataEZC(1,1).Price
```

```
ans =
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [   660.5000]
    []
    []
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [   660.5000]
    [-2.1475e+09]
```

Close the CQG Connection

```
close(c)
```

See Also

`cqg` | `close` | `createOrder` | `history` | `timeseries` | `startUp` | `shutDown` | `realtime`

Related Examples

- “Create Order Using CQG” on page 3-18

- “Create CQG Orders” on page 3-20
- “Request CQG Historical Data” on page 3-24
- “Request CQG Intraday Tick Data” on page 3-27

More About

- “Workflow for CQG” on page 3-16

External Websites

- *CQG API Reference Guide*

Bloomberg EMSX Topics

Create Order Using Bloomberg EMSX

This example shows how to connect to Bloomberg EMSX and create and route a market order.

For details about connecting to Bloomberg EMSX and creating orders, see the *EMSX API Programmer's Guide*.

Connect to Bloomberg EMSX

1. If you are using `emsx` for the first time, install a Java archive file from Bloomberg for `emsx` and other Bloomberg commands to work correctly.

   If you already have `blpapi3.jar` downloaded from Bloomberg, you can find it in your Bloomberg folders at `..\blp\api\APIv3\JavaAPI\lib\blpapi3.jar` or `..\blp\api\APIv3\JavaAPI\v3.x\lib\blpapi3.jar`. If you have `blpapi3.jar`, go to step 3.

   If `blpapi3.jar` is not downloaded from Bloomberg, then download it as follows:

   a. In your Bloomberg terminal, type `WAPI {GO}` to open the API Developer's Help Site screen.
   b. Click API Download Center, then click Desktop API.
   c. After downloading `blpapi3.jar` on your system, add it to the MATLAB Java class path using the `javaaddpath` function.

      Execute the `javaaddpath` function for every session of MATLAB. To avoid executing the `javaaddpath` function at every session, add `javaaddpath` to your `startup.m` file or add the full path for `blpapi3.jar` to your `javaclasspath.txt` file. For details about `javaclasspath.txt`, see "Java Class Path". For details about editing your `startup.m` file, see "Startup Options in MATLAB Startup File".

2. Connect to the Bloomberg EMSX test service.

```
c = emsx('//blp/emapisvc_beta')
```

```
c =

  emsx with properties:

      Session: [1x1 com.bloomberglp.blpapi.Session]
      Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
    Ipaddress: 'localhost'
         Port: 8194
```

   MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following:

   - Bloomberg EMSX session object
   - Bloomberg EMSX service object
   - IP address of the machine running the Bloomberg EMSX test service
   - Port number of the machine running the Bloomberg EMSX test service

Create Market Order Request

Create an order request structure `order` for a buy market order of 400 shares of IBM. Specify the broker as `EFIX`, use any hand instruction, and set the time in force to `DAY`.

```
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_SIDE = 'BUY';
```

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(400);
order.EMSX_BROKER = 'EFIX';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_TIF = 'DAY';
```

Create and Route Market Order

Create and route the market order using the Bloomberg EMSX connection `c` and order request structure `order`.

```
events = createOrderAndRoute(c,order)
```

```
events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Close Bloomberg EMSX Connection

```
close(c)
```

See Also

`emsx` | `createOrderAndRoute` | `close`

Related Examples

- “Create and Manage Bloomberg EMSX Order” on page 4-4
- “Create and Manage Bloomberg EMSX Route” on page 4-8
- “Manage Bloomberg EMSX Order and Route” on page 4-12

More About

- “Workflow for Bloomberg EMSX” on page 4-16

External Websites

- *EMSX API Programmers Guide*

Create and Manage Bloomberg EMSX Order

This example shows how to connect to Bloomberg EMSX, create an order, and interact with the order.

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Connect to Bloomberg EMSX

Connect to the Bloomberg EMSX test service. Display the current event queue contents using `processEvent`.

```
c = emsx('//blp/emapisvc_beta');
processEvent(c)
```

```
c =

  emsx with properties:

      Session: [1x1 com.bloomberglp.blpapi.Session]
      Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
    Ipaddress: 'localhost'
         Port: 8194

SessionConnectionUp = {
    server = localhost/127.0.0.1:8194
}

SessionStarted = {
}

ServiceOpened = {
    serviceName = //blp/emapisvc_beta
}
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service

`processEvent` displays events associated with connecting to Bloomberg EMSX.

Set Up Order Subscription

Subscribe to order events using the Bloomberg EMSX connection `c` associated with these Bloomberg EMSX fields.

```
fields = {'EMSX_TICKER','EMSX_AMOUNT','EMSX_FILL'};

[events,subs] = orders(c,fields)
```

```
events =

                        MSG_TYPE: {'E'}
```

```
                    MSG_SUB_TYPE: {'O'}
                    EVENT_STATUS: 4
                    ...

subs =

com.bloomberglp.blpapi.SubscriptionList@4bc3dc78
```

`events` contains fields for the events associated with the existing Bloomberg EMSX orders. `subs` contains the Bloomberg EMSX subscription list object.

Create Order

Create an order request structure `order` for a buy market order of 400 shares of IBM. Specify the broker as `EFIX`, use any hand instruction, and set the time in force to `DAY`.

```
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_SIDE = 'BUY';
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(400);
order.EMSX_BROKER = 'EFIX';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_TIF = 'DAY';
```

Create the order using the Bloomberg EMSX connection `c` and the order request structure `order`.

```
events = createOrder(c,order)
```

```
order_events =

        EMSX_SEQUENCE: 354646
              MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Modify Order

Define the structure `modorder` that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`

This code modifies order number `354646` for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(354646);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Modify the order using the Bloomberg EMSX connection `c` and modify order structure `modorder`.

```
events = modifyOrder(c,modorder)
```

```
events =

    EMSX_SEQUENCE: 354646
          MESSAGE: 'Order Modified'
```

The default event handler processes the events associated with modifying an order. `modifyOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Delete Order

Define the structure `ordernum` that contains the order sequence number `354646` for the order to delete. Delete the order using the Bloomberg EMSX connection `c` and the delete order number structure `ordernum`.

```
ordernum.EMSX_SEQUENCE = 354646;

events = deleteOrder(c,ordernum)
```

```
events =

      STATUS: '0'
     MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting an order. `deleteOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Stop Order Subscription

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`.

```
c.Session.unsubscribe(subs)
```

Close Bloomberg EMSX Connection

```
close(c)
```

See Also

`emsx` | `close` | `createOrder` | `orders` | `modifyOrder` | `deleteOrder`

Related Examples

- “Create Order Using Bloomberg EMSX” on page 4-2
- “Create and Manage Bloomberg EMSX Route” on page 4-8
- “Manage Bloomberg EMSX Order and Route” on page 4-12

More About

- “Workflow for Bloomberg EMSX” on page 4-16

External Websites

- *EMSX API Programmers Guide*

Create and Manage Bloomberg EMSX Route

This example shows how to connect to Bloomberg EMSX, set up a route subscription, create and route an order, and interact with the route.

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Connect to Bloomberg EMSX

Connect to the Bloomberg EMSX test service. Display the current event queue contents using `processEvent`.

```
c = emsx('//blp/emapisvc_beta');
processEvent(c)
```

```
c =

  emsx with properties:

      Session: [1x1 com.bloomberglp.blpapi.Session]
      Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
    Ipaddress: 'localhost'
         Port: 8194

SessionConnectionUp = {
    server = localhost/127.0.0.1:8194
}

SessionStarted = {
}

ServiceOpened = {
    serviceName = //blp/emapisvc_beta
}
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service

`processEvent` displays events associated with connecting to Bloomberg EMSX.

Set Up Route Subscription

Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`. Return the status for existing routes.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};

[events,subs] = routes(c,fields)
```

```
events =
```

```
                    MSG_TYPE: {5x1 cell}
                MSG_SUB_TYPE: {5x1 cell}
                EVENT_STATUS: [5x1 int32]
                ...

subs =

com.bloomberglp.blpapi.SubscriptionList@463b9287
```

`events` contains fields for the events currently in the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Create and Route Order

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection `c` and the order request structure `order`.

```
events = createOrderAndRoute(c,order)

events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Modify Route

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 50 shares of IBM for order sequence number `335877` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(335877)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(50);
modroute.EMSX_ROUTE_ID = int32(1);
```

Modify the route using the Bloomberg EMSX connection `c` and modify route request `modroute`.

```
events = modifyRoute(c,modroute)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Delete Modified Route

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` and the route number `EMSX_ROUTE_ID` associated with the modified route.

```
routenum.EMSX_SEQUENCE = 0;
routenum.EMSX_ROUTE_ID = 0;
```

Delete the route using the Bloomberg EMSX connection `c` and delete route number structure `routenum`.

```
events = deleteRoute(c,routenum)
```

```
events =

     STATUS: '1'
    MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting a route. `deleteRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Stop Route Subscription

Unsubscribe from route events using the Bloomberg EMSX subscription list object `subs`.

```
c.Session.unsubscribe(subs)
```

Close Bloomberg EMSX Connection

```
close(c)
```

See Also

`emsx` | `close` | `createOrderAndRoute` | `modifyRoute` | `deleteRoute` | `routes` | `routeOrder`

Related Examples

- “Create Order Using Bloomberg EMSX” on page 4-2
- “Create and Manage Bloomberg EMSX Order” on page 4-4
- “Manage Bloomberg EMSX Order and Route” on page 4-12

More About

- “Workflow for Bloomberg EMSX” on page 4-16

External Websites

- *EMSX API Programmers Guide*

Manage Bloomberg EMSX Order and Route

This example shows how to connect to Bloomberg EMSX, set up an order and route subscription, create and route an order, and interact with the route.

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Connect to Bloomberg EMSX

Connect to the Bloomberg EMSX test service. Display the current event queue contents using `processEvent`.

```
c = emsx('//blp/emapisvc_beta');
processEvent(c)
```

```
c =

  emsx with properties:

      Session: [1x1 com.bloomberglp.blpapi.Session]
      Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
    Ipaddress: 'localhost'
         Port: 8194

SessionConnectionUp = {
    server = localhost/127.0.0.1:8194
}

SessionStarted = {
}

ServiceOpened = {
    serviceName = //blp/emapisvc_beta
}
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service

`processEvent` displays events associated with connecting to Bloomberg EMSX.

Set Up Order and Route Subscription

Subscribe to order events using the Bloomberg EMSX connection `c` associated with these Bloomberg EMSX fields.

```
fields = {'EMSX_TICKER','EMSX_AMOUNT','EMSX_FILL'};

[events,osubs] = orders(c,fields)
```

```
events =
```

```
                    MSG_TYPE: {'E'}
                MSG_SUB_TYPE: {'O'}
                EVENT_STATUS: 4
                ...

osubs =

com.bloomberglp.blpapi.SubscriptionList@4bc3dc78
```

`events` contains fields for the events associated with the existing Bloomberg EMSX orders. `osubs` contains the Bloomberg EMSX subscription list object.

Subscribe to route events for the Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`. Return the status for existing routes.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};

[events,rsubs] = routes(c,fields)

events =

                    MSG_TYPE: {5x1 cell}
                MSG_SUB_TYPE: {5x1 cell}
                EVENT_STATUS: [5x1 int32]
                ...

rsubs =

com.bloomberglp.blpapi.SubscriptionList@463b9287
```

`events` contains fields for the events currently in the event queue. `rsubs` contains the Bloomberg EMSX subscription list object.

Create and Route Order

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection `c` and the order request structure `order`.

```
events = createOrderAndRoute(c,order)

events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Modify Route

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 50 shares of IBM for order sequence number `335877` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(335877)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(50);
modroute.EMSX_ROUTE_ID = int32(1);
```

Modify the route using the Bloomberg EMSX connection `c` and modify route request `modroute`.

```
events = modifyRoute(c,modroute)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Delete Route

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 0;
routenum.EMSX_ROUTE_ID = 0;
```

Delete the route using the Bloomberg EMSX connection `c` and delete route number structure `routenum`.

```
events = deleteRoute(c,routenum)
```

```
events =
```

```
     STATUS: '1'
    MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting a route. `deleteRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Stop Order and Route Subscription

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close Bloomberg EMSX Connection

```
close(c)
```

See Also

`emsx` | `close` | `createOrderAndRoute` | `orders` | `modifyRoute` | `deleteRoute` | `routes`

Related Examples

- “Create Order Using Bloomberg EMSX” on page 4-2
- “Create and Manage Bloomberg EMSX Order” on page 4-4
- “Create and Manage Bloomberg EMSX Route” on page 4-8

More About

- “Workflow for Bloomberg EMSX” on page 4-16

External Websites

- *EMSX API Programmers Guide*

Workflow for Bloomberg EMSX

The workflow for Bloomberg EMSX is versatile with many options for alternate flows to create, route, and manage the status of an open order until it is filled.

1. Connect to Bloomberg EMSX using `emsx`.
2. Set up a subscription for orders and routes to obtain events on subsequent requests using `orders` and `routes`.
3. Create a Bloomberg EMSX order by completing one or more of these steps:
   - Create an order using `createOrder`.
   - Route an order using `routeOrder`.
   - Route an order with strategies using `routeOrderWithStrat`.
   - Route a group of orders using `groupRouteOrder`.
   - Route a group of orders with strategies using `groupRouteOrderWithStrat`.
   - Create an order and route using `createOrderAndRoute`.
   - Create an order and route with strategies using `createOrderAndRouteWithStrat`.
   - Create a basket of orders using `createBasket`.
   - Manually fill an order using `manualFill`.
4. Modify an order or route using these functions:
   - Modify an order using `modifyOrder`.
   - Modify a route using `modifyRoute`.
   - Modify a route with a strategy using `modifyRouteWithStrat`.
5. Delete an order or route using these functions:
   - Delete an order using `deleteOrder`.
   - Delete a route using `deleteRoute`.
6. Obtain information from Bloomberg EMSX using these functions:
   - Obtain broker information using `getBrokerInfo`.
   - Obtain Bloomberg EMSX field information using `getAllFieldMetaData`.
7. Explore information about existing orders and routes using these functions:
   - View order transactions with a sample order blotter using `emsxOrderBlotter`.
   - Process the current contents of the event queue using `processEvent`.
8. Close the Bloomberg EMSX connection using `close`.

See Also

Related Examples

- “Create Order Using Bloomberg EMSX” on page 4-2
- “Create and Manage Bloomberg EMSX Order” on page 4-4
- “Create and Manage Bloomberg EMSX Route” on page 4-8

- “Manage Bloomberg EMSX Order and Route” on page 4-12

External Websites

- *EMSX API Programmers Guide*

5

Topics for Bloomberg C++ Interfaces

Create Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface

This example shows how to connect to Bloomberg EMSX and create and route a market order using the Bloomberg EMSX C++ interface.

For details about connecting to Bloomberg EMSX and creating orders, see the *EMSX API Programmer's Guide*.

Connect to Bloomberg EMSX

Connect to the Bloomberg EMSX test service using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

```
c =

  bloombergEMSX with properties:

    Session: [1×1 datafeed.internal.BLPSession]
    Service: '//blp/emapisvc_beta'
    Ipaddress: "111.222.333.44"
    Port: 8194.00
    User: []
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service

Create Market Order Request

Create an order request structure `order` for a buy market order of 400 shares of IBM. Specify the broker as `EFIX`, use any hand instruction, and set the time in force to `DAY`.

```
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_SIDE = 'BUY';
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(400);
order.EMSX_BROKER = 'EFIX';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_TIF = 'DAY';
```

Create and Route Market Order

Create and route the market order using the Bloomberg EMSX connection `c` and order request structure `order`.

```
events = createOrderAndRoute(c,order)
```

```
events =

    EMSX_SEQUENCE: 335877
```

```
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Close Bloomberg EMSX Connection

```
close(c)
```

See Also

Objects
`bloombergEMSX`

Functions
`close` | `createOrderAndRoute`

More About

- “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
- “Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
- “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11

External Websites

- *EMSX API Programmers Guide*

Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface

This example shows how to connect to Bloomberg EMSX with the Bloomberg EMSX C++ interface, create an order, and interact with the order.

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Connect to Bloomberg EMSX

Connect to the Bloomberg EMSX test service using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

```
c =

  bloombergEMSX with properties:

    Session: [1×1 datafeed.internal.BLPSession]
    Service: '//blp/emapisvc_beta'
    Ipaddress: "111.222.333.44"
    Port: 8194.00
    User: []
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service

Set Up Order Subscription

Subscribe to order events using the Bloomberg EMSX connection `c` associated with these Bloomberg EMSX fields.

```
fields = {'EMSX_TICKER','EMSX_AMOUNT','EMSX_FILL'};
```

```
events = orders(c,fields)
```

```
events =

                    MSG_TYPE: {'E'}
                MSG_SUB_TYPE: {'O'}
                EVENT_STATUS: 4
                ...
```

`events` contains fields for the events associated with the existing Bloomberg EMSX orders.

Create Order

Create an order request structure `order` for a buy market order of 400 shares of IBM. Specify the broker as `EFIX`, use any hand instruction, and set the time in force to `DAY`.

```
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_SIDE = 'BUY';
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(400);
order.EMSX_BROKER = 'EFIX';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_TIF = 'DAY';
```

Create the order using the Bloomberg EMSX connection `c` and the order request structure `order`.

```
events = createOrder(c,order)
```

```
order_events =

        EMSX_SEQUENCE: 354646
              MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Modify Order

Define the structure `modorder` that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`

This code modifies order number `354646` for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(354646);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Modify the order using the Bloomberg EMSX connection `c` and modify order structure `modorder`.

```
events = modifyOrder(c,modorder)
```

```
events =

    EMSX_SEQUENCE: 354646
          MESSAGE: 'Order Modified'
```

The default event handler processes the events associated with modifying an order. `modifyOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Delete Order

Define the structure `ordernum` that contains the order sequence number `354646` for the order to delete. Delete the order using the Bloomberg EMSX connection `c` and the delete order number structure `ordernum`.

```
ordernum.EMSX_SEQUENCE = 354646;

events = deleteOrder(c,ordernum)

events =

      STATUS: '0'
     MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting an order. `deleteOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Stop Order Subscription

Unsubscribe from order events using the Bloomberg EMSX subscriptions.

```
c.Session.stopSubscriptions
```

Close Bloomberg EMSX Connection

```
close(c)
```

See Also

Objects
`bloombergEMSX`

Functions
`orders` | `close` | `createOrder` | `modifyOrder` | `deleteOrder`

More About

- “Create Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-2
- “Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
- “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11

External Websites

- *EMSX API Programmers Guide*

Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface

This example shows how to connect to Bloomberg EMSX with the Bloomberg EMSX C++ interface, set up a route subscription, create and route an order, and interact with the route.

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Connect to Bloomberg EMSX

Connect to the Bloomberg EMSX test service using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

```
c =

  bloombergEMSX with properties:

    Session: [1×1 datafeed.internal.BLPSession]
    Service: '//blp/emapisvc_beta'
    Ipaddress: "111.222.333.44"
    Port: 8194.00
    User: []
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service

Set Up Route Subscription

Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`. Return the status for existing routes.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};
```

```
events = routes(c,fields)
```

```
events =

                    MSG_TYPE: {5x1 cell}
                MSG_SUB_TYPE: {5x1 cell}
                EVENT_STATUS: [5x1 int32]
                ...
```

`events` contains fields for the events currently in the event queue.

Create and Route Order

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection `c` and the order request structure `order`.

```
events = createOrderAndRoute(c,order)
```

```
events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Modify Route

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 50 shares of IBM for order sequence number `335877` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(335877)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(50);
modroute.EMSX_ROUTE_ID = int32(1);
```

Modify the route using the Bloomberg EMSX connection `c` and modify route request `modroute`.

```
events = modifyRoute(c,modroute)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Delete Modified Route

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` and the route number `EMSX_ROUTE_ID` associated with the modified route.

```
routenum.EMSX_SEQUENCE = 0;
routenum.EMSX_ROUTE_ID = 0;
```

Delete the route using the Bloomberg EMSX connection `c` and delete route number structure `routenum`.

```
events = deleteRoute(c,routenum)
```

```
events =

     STATUS: '1'
    MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting a route. `deleteRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Stop Route Subscription

Unsubscribe from route events using the Bloomberg EMSX subscriptions.

```
c.Session.stopSubscriptions
```

Close Bloomberg EMSX Connection

```
close(c)
```

See Also

Objects
`bloombergEMSX`

Functions
`routes` | `close` | `createOrderAndRoute` | `modifyRoute` | `deleteRoute`

More About

- “Create Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-2
- “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
- “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11

External Websites

- *EMSX API Programmers Guide*

Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface

This example shows how to connect to Bloomberg EMSX with the Bloomberg EMSX C++ interface, set up an order and route subscription, create and route an order, and interact with the route.

For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Connect to Bloomberg EMSX

Connect to the Bloomberg EMSX test service using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

```
c =

  bloombergEMSX with properties:

    Session: [1×1 datafeed.internal.BLPSession]
    Service: '//blp/emapisvc_beta'
    Ipaddress: "111.222.333.44"
    Port: 8194.00
    User: []
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service

Set Up Order and Route Subscription

Subscribe to order events using the Bloomberg EMSX connection `c` associated with these Bloomberg EMSX fields.

```
fields = {'EMSX_TICKER','EMSX_AMOUNT','EMSX_FILL'};
```

```
events = orders(c,fields)
```

```
events =

                    MSG_TYPE: {'E'}
                MSG_SUB_TYPE: {'O'}
                EVENT_STATUS: 4
                ...
```

`events` contains fields for the events associated with the existing Bloomberg EMSX orders.

Subscribe to route events for the Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`. Return the status for existing routes.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};

events = routes(c,fields)

events =

                    MSG_TYPE: {5x1 cell}
                MSG_SUB_TYPE: {5x1 cell}
                EVENT_STATUS: [5x1 int32]
                ...
```

`events` contains fields for the events currently in the event queue.

Create and Route Order

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection `c` and the order request structure `order`.

```
events = createOrderAndRoute(c,order)

events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Modify Route

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 50 shares of IBM for order sequence number `335877` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(335877)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(50);
modroute.EMSX_ROUTE_ID = int32(1);
```

Modify the route using the Bloomberg EMSX connection `c` and modify route request `modroute`.

```
events = modifyRoute(c,modroute)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Delete Route

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 0;
routenum.EMSX_ROUTE_ID = 0;
```

Delete the route using the Bloomberg EMSX connection `c` and delete route number structure `routenum`.

```
events = deleteRoute(c,routenum)
```

```
events =

     STATUS: '1'
    MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting a route. `deleteRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Stop Order and Route Subscription

Unsubscribe from order and route events using the Bloomberg EMSX subscriptions.

```
c.Session.stopSubscriptions
```

Close Bloomberg EMSX Connection

```
close(c)
```

See Also

Objects
`bloombergEMSX`

Functions
`orders` | `routes` | `close` | `createOrderAndRoute` | `modifyRoute` | `deleteRoute`

More About

- “Create Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-2
- “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
- “Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7

External Websites

- *EMSX API Programmers Guide*

Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface

This example shows how to connect to Bloomberg and retrieve current and historical Bloomberg market data. For details about Bloomberg connection requirements, see “Data Server Connection Requirements” on page 1-3 and “Installing Bloomberg and Configuring Connections” on page 1-5.

Connect to Bloomberg

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Retrieve Current Data

Format MATLAB data display for currency.

```
format bank
```

Retrieve closing and open prices for Microsoft.

```
sec = 'MSFT US Equity';
fields = {'LAST_PRICE';'OPEN'}; % closing and open prices

[d,sec] = getdata(c,sec,fields)
```

```
d =

  struct with fields:

    LAST_PRICE: 62.32
          OPEN: 62.48


sec =

  cell

    'MSFT US Equity'
```

`d` contains the Bloomberg closing and open prices. `sec` contains the Bloomberg security name for Microsoft.

Retrieve Historical Data

Retrieve monthly closing and open price data from January 1, 2012, through December 31, 2012, for Microsoft.

```
fromdate = '1/01/2012'; % beginning of date range for historical data
todate = '12/31/2012'; % ending of date range for historical data
period = 'monthly'; % retrieve monthly data

[d,sec] = history(c,sec,fields,fromdate,todate,period)


d =

     734899.00         29.53         26.55
     734928.00         31.74         29.79
     734959.00         32.26         31.93
     734989.00         32.02         32.22
     735020.00         29.19         32.05
     735050.00         30.59         28.76
     735081.00         29.47         30.62
     735112.00         30.82         29.59
     735142.00         29.76         30.45
     735173.00         28.54         29.81
     735203.00         26.61         28.84
     735234.00         26.71         26.78


sec =

  cell

    'MSFT US Equity'
```

`d` contains the numeric representation of the date in the first column, closing price in the second column, and open price in the third column. Each row represents data for one month in the date range. `sec` contains the Bloomberg security name for Microsoft.

Find Maximum Open Price in Date Range

Calculate the maximum open price for the year 2012.

```
openprices = d(:,3); % retrieve all open prices in date range
max(openprices) % calculate maximum open price


ans =

         32.22
```

Close Bloomberg Connection

```
close(c)
```

See Also

Objects
`bloomberg`

Functions
`isconnection` | `getdata` | `history` | `close`

Related Examples

- “Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface” on page 5-18
- “Retrieve Bloomberg Historical Data Using Bloomberg Desktop C++ Interface” on page 5-20
- “Retrieve Bloomberg Intraday Tick Data Using Bloomberg Desktop C++ Interface” on page 5-24
- “Retrieve Bloomberg Real-Time Data Using Bloomberg Desktop C++ Interface” on page 5-26

Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface

This example shows how to retrieve current data from Bloomberg for a single security and for multiple securities.

Connect to Bloomberg

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Retrieve Current Data for Single Security

Retrieve last and open prices for Microsoft.

`d` contains the Bloomberg last and open prices as fields in a structure. `sec` contains the Bloomberg security name for Microsoft in a cell array. The security name is a character vector.

```
sec = 'MSFT US Equity';
fields = {'LAST_PRICE';'OPEN'}; % Retrieve data for last and open prices

[d,sec] = getdata(c,sec,fields)
```

```
d =

  struct with fields:

    LAST_PRICE: 62.30
          OPEN: 62.95


sec =

  cell

    'MSFT US Equity'
```

Retrieve Current Data for Multiple Securities

Retrieve last and open prices for the IBM and Ford Motor Company securities.

`d` contains the Bloomberg last and open prices as fields in a structure. `sec` contains the Bloomberg security names for IBM and Ford Motor Company in a cell array. Each security name is a character vector.

```
s = {'IBM US Equity','F US Equity'};
fields = {'LAST_PRICE';'OPEN'}; % Retrieve data for last and open prices
```

```
[d,sec] = getdata(c,s,fields)

d =

  struct with fields:

    LAST_PRICE: [2×1 double]
          OPEN: [2×1 double]


sec =

  2×1 cell array

    'IBM US Equity'
    'F US Equity'
```

Display the last price for both securities.

```
d.LAST_PRICE

ans =

        166.73
         12.63
```

Close Bloomberg Connection

```
close(c)
```

See Also

Objects
`bloomberg`

Functions
`isconnection` | `getdata` | `close`

Related Examples

- “Retrieve Bloomberg Historical Data Using Bloomberg Desktop C++ Interface” on page 5-20
- “Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15
- “Retrieve Bloomberg Intraday Tick Data Using Bloomberg Desktop C++ Interface” on page 5-24
- “Retrieve Bloomberg Real-Time Data Using Bloomberg Desktop C++ Interface” on page 5-26

Retrieve Bloomberg Historical Data Using Bloomberg Desktop C++ Interface

This example shows how to retrieve historical data from Bloomberg for a single security. The example shows retrieving weekly data within a date range and retrieving data with a default period. Then, the example also shows how to retrieve data for multiple securities.

Connect to Bloomberg

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Retrieve Historical Data for One Security

Retrieve monthly closing and open price data from January 1, 2012, through December 31, 2012, for Microsoft.

```
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
fromdate = '1/01/2012';
todate = '12/31/2012';
period = 'monthly';

[d,sec] = history(c,s,f,fromdate,todate,period)
```

```
d =

     734899.00         27.87         25.06
     734928.00         30.16         28.12
     734959.00         30.65         30.34
     ...

sec =

  cell

    'MSFT US Equity'
```

`d` contains the numeric representation of the date in the first column, closing price in the second column, and open price in the third column. Each row represents data for one month in the date range. `sec` contains the Bloomberg security name for Microsoft.

Retrieve Weekly Historical Data

Retrieve the weekly closing prices from November 1, 2010, through December 23, 2010, for the Microsoft security using US currency. In this case, the anchor date depends on the date December 23, 2010. Because this date is a Thursday, each previous value is reported for the Thursday of the week in question.

```
f = 'LAST_PRICE';
fromdate = '11/01/2010';
todate =  '12/23/2010';
period = {'weekly'};
currency = 'USD';

[d,sec] = history(c,s,f,fromdate,todate, ...
    period,currency)

d =

     734446.00         27.14
     734453.00         26.68
     734460.00         25.84
     734467.00         25.37
     734474.00         26.89
     734481.00         27.08
     734488.00         27.99
     734495.00         28.30


sec =

  1×1 cell array

    {'MSFT US Equity'}
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the Microsoft security.

Retrieve Historical Data Using Default Period

Retrieve the closing prices from August 1, 2010, through September 10, 2010, for the Microsoft security in US currency, and set the default period of the data by using `[]`. The default period of a security depends on the security itself.

```
fromdate = '8/01/2010';
todate = '9/10/2010';
currency = 'USD';

[d,sec] = history(c,s,f,fromdate,todate, ...
    [],currency)

d =

     734352.00         26.33
     734353.00         26.16
     734354.00         25.73
     ...

sec =

  1×1 cell array

    {'MSFT US Equity'}
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the Microsoft security.

Retrieve Historical Data for Multiple Securities

Retrieve monthly closing and open prices from January 1, 2012, through December 31, 2012, for the IBM and Ford Motor Company securities.

`d` is a cell array of double matrices that contains the historical data for both securities. `sec` contains the Bloomberg security names for the IBM and Ford Motor Company securities in a cell array. Each security name is a character vector.

```
s = {'IBM US Equity','F US Equity'};
f = {'LAST_PRICE';'OPEN'};
fromdate = '1/01/2012';
todate = '12/31/2012';
period = 'monthly';

[d,sec] = history(c,s,f,fromdate,todate,period)
```

```
d =

  2×1 cell array

    [12×3 double]
    [12×3 double]


sec =

  2×1 cell array

    'IBM US Equity'
    'F US Equity'
```

Display the closing and open prices for the first security.

```
d{1}
```

```
ans =

     734899.00        192.60        186.73
     734928.00        196.73        193.21
     734959.00        208.65        197.23
     ...
```

The data in the double matrix is:

- First column — Numeric representation of the date
- Second column — Closing price
- Third column — Open price

Each row represents data for one month in the date range.

Close Bloomberg Connection

```
close(c)
```

See Also

Objects
`bloomberg`

Functions
`isconnection` | `history` | `close`

Related Examples

- “Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface” on page 5-18
- “Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15
- “Retrieve Bloomberg Intraday Tick Data Using Bloomberg Desktop C++ Interface” on page 5-24
- “Retrieve Bloomberg Real-Time Data Using Bloomberg Desktop C++ Interface” on page 5-26

Retrieve Bloomberg Intraday Tick Data Using Bloomberg Desktop C++ Interface

This example shows how to retrieve intraday tick data from Bloomberg.

Connect to Bloomberg

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Retrieve Intraday Tick Data

Retrieve the trade tick series for the past 50 days for the IBM security aggregated into 5-minute intervals.

```
d = timeseries(c,'IBM US Equity',{floor(now)-50,floor(now)},5,'Trade')
```

```
ans =

  Columns 1 through 7

   735487.40        187.20        187.60        187.02        187.08     207683.00        560.00
   735487.40        187.03        187.13        186.65        186.78      46990.00        349.00
   735487.40        186.78        186.78        186.40        186.47      51589.00        399.00
     ...

Column 8

   38902968.00
    8779374.00
    9626896.00
   ...
```

The columns in `d` contain the following:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row of data shows prices and tick data for the current date. The next row shows tick data for 5 minutes later.

Close Bloomberg Connection

```
close(c)
```

See Also

Objects
`bloomberg`

Functions
`timeseries` | `isconnection` | `close`

Related Examples

- “Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface” on page 5-18
- “Retrieve Bloomberg Historical Data Using Bloomberg Desktop C++ Interface” on page 5-20
- “Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15
- “Retrieve Bloomberg Real-Time Data Using Bloomberg Desktop C++ Interface” on page 5-26

Retrieve Bloomberg Real-Time Data Using Bloomberg Desktop C++ Interface

This example shows how to retrieve real-time data from Bloomberg. Here, to display Bloomberg stock tick data at the command line, use the event handler `disp`. Instead of the default event handler, you can create your own event handler function to process Bloomberg data.

Connect to Bloomberg

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Retrieve Real-Time Data

Retrieve the last price and volume for IBM and Ford Motor Company securities.

```
[~,t] = realtime(c,{'IBM US Equity','F US Equity'}, ...
    {'LAST_PRICE','VOLUME'},'disp')
```

```
t =


   Timer Object: timer-4

   Timer Settings
      ExecutionMode: fixedRate
              Period: 0.05
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: 1x5 cell array
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''

Columns 1 through 6

    {'SecurityID' }    {'LAST_PRICE'}    {'SecurityID' }    {'VOLUME'  }    {'SecurityID'   }
    {'F US Equity'}    {'8.960000'  }    {'F US Equity'}    {'13423731'}    {'IBM US Equity'}

  Columns 7 through 8

    {'SecurityID'   }    {'VOLUME'}
    {'IBM US Equity'}    {'744066'}
...
```

`realtime` returns the MATLAB timer object with its properties. Then, `realtime` returns the stock tick data for the IBM and Ford Motor Company securities with the last price and volume.

Stop the display of real-time data.

```
stop(t)
c.Session.stopSubscriptions
```

Close Bloomberg Connection

```
close(c)
```

See Also

Objects
`bloomberg`

Functions
`isconnection` | `realtime` | `close`

Related Examples

- “Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface” on page 5-18
- “Retrieve Bloomberg Historical Data Using Bloomberg Desktop C++ Interface” on page 5-20
- “Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15
- “Retrieve Bloomberg Intraday Tick Data Using Bloomberg Desktop C++ Interface” on page 5-24
- “Writing and Running Custom Event Handler Functions” on page 1-26

Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface

This example shows how to retrieve current data from Bloomberg for a single security and for multiple securities.

Connect to Bloomberg

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Retrieve Current Data for Single Security

Retrieve last and open prices for Microsoft.

`d` contains the Bloomberg last and open prices as fields in a structure. `sec` contains the Bloomberg security name for Microsoft in a cell array. The security name is a character vector.

```
sec = 'MSFT US Equity';
fields = {'LAST_PRICE';'OPEN'}; % Retrieve data for last and open prices

[d,sec] = getdata(c,sec,fields)
```

```
d =

  struct with fields:

    LAST_PRICE: 62.30
          OPEN: 62.95
```

```
sec =

  cell

    'MSFT US Equity'
```

Retrieve Current Data for Multiple Securities

Retrieve last and open prices for the IBM and Ford Motor Company securities.

`d` contains the Bloomberg last and open prices as fields in a structure. `sec` contains the Bloomberg security names for IBM and Ford Motor Company in a cell array. Each security name is a character vector.

```
s = {'IBM US Equity','F US Equity'};
fields = {'LAST_PRICE';'OPEN'}; % Retrieve data for last and open prices

[d,sec] = getdata(c,s,fields)
```

```
d =

  struct with fields:

    LAST_PRICE: [2×1 double]
          OPEN: [2×1 double]


sec =

  2×1 cell array

    'IBM US Equity'
    'F US Equity'
```

Display the last price for both securities.

```
d.LAST_PRICE
```

```
ans =

        166.73
         12.63
```

Close Bloomberg Connection

```
close(c)
```

See Also

Objects
`bloombergBPIPE`

Functions
`close` | `getdata` | `isconnection`

Related Examples

- “Retrieve Bloomberg Historical Data Using Bloomberg B-PIPE C++ Interface” on page 5-31
- “Retrieve Bloomberg Intraday Tick Data Using Bloomberg B-PIPE C++ Interface” on page 5-35
- “Retrieve Bloomberg Real-Time Data Using Bloomberg B-PIPE C++ Interface” on page 5-37

Retrieve Bloomberg Historical Data Using Bloomberg B-PIPE C ++ Interface

This example shows how to retrieve historical data from Bloomberg for a single security. The example shows retrieving weekly data within a date range and retrieving data with a default period. Then, the example also shows how to retrieve data for multiple securities.

Connect to Bloomberg

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Retrieve Historical Data for One Security

Retrieve monthly closing and open price data from January 1, 2012, through December 31, 2012, for Microsoft.

```
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
fromdate = '1/01/2012';
todate = '12/31/2012';
period = 'monthly';

[d,sec] = history(c,s,f,fromdate,todate,period)
```

```
d =

     734899.00         27.87         25.06
     734928.00         30.16         28.12
     734959.00         30.65         30.34
     ...

sec =
```

```
  cell

    'MSFT US Equity'
```

d contains the numeric representation of the date in the first column, closing price in the second column, and open price in the third column. Each row represents data for one month in the date range. sec contains the Bloomberg security name for Microsoft.

Retrieve Weekly Historical Data

Retrieve the weekly closing prices from November 1, 2010, through December 23, 2010, for the Microsoft security using US currency. In this case, the anchor date depends on the date December 23, 2010. Because this date is a Thursday, each previous value is reported for the Thursday of the week in question.

```
f = 'LAST_PRICE';
fromdate = '11/01/2010';
todate =  '12/23/2010';
period = {'weekly'};
currency = 'USD';

[d,sec] = history(c,s,f,fromdate,todate, ...
    period,currency)
```

```
d =

     734446.00         27.14
     734453.00         26.68
     734460.00         25.84
     734467.00         25.37
     734474.00         26.89
     734481.00         27.08
     734488.00         27.99
     734495.00         28.30


sec =

  1×1 cell array

    {'MSFT US Equity'}
```

d contains the numeric representation for the date in the first column and the closing price in the second column. sec contains the name of the Microsoft security.

Retrieve Historical Data Using Default Period

Retrieve the closing prices from August 1, 2010, through September 10, 2010, for the Microsoft security in US currency, and set the default period of the data by using []. The default period of a security depends on the security itself.

```
fromdate = '8/01/2010';
todate = '9/10/2010';
currency = 'USD';

[d,sec] = history(c,s,f,fromdate,todate, ...
    [],currency)
```

```
d =

     734352.00         26.33
     734353.00         26.16
     734354.00         25.73
     ...

sec =

  1×1 cell array

    {'MSFT US Equity'}
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the Microsoft security.

Retrieve Historical Data for Multiple Securities

Retrieve monthly closing and open prices from January 1, 2012, through December 31, 2012, for the IBM and Ford Motor Company securities.

`d` is a cell array of double matrices that contains the historical data for both securities. `sec` contains the Bloomberg security names for the IBM and Ford Motor Company securities in a cell array. Each security name is a character vector.

```
s = {'IBM US Equity','F US Equity'};
f = {'LAST_PRICE';'OPEN'};
fromdate = '1/01/2012';
todate = '12/31/2012';
period = 'monthly';

[d,sec] = history(c,s,f,fromdate,todate,period)

d =

  2×1 cell array

    [12×3 double]
    [12×3 double]


sec =

  2×1 cell array

    'IBM US Equity'
    'F US Equity'
```

Display the closing and open prices for the first security.

```
d{1}

ans =

     734899.00        192.60        186.73
     734928.00        196.73        193.21
     734959.00        208.65        197.23
     ...
```

The data in the double matrix is:

- First column — Numeric representation of the date
- Second column — Closing price
- Third column — Open price

Each row represents data for one month in the date range.

Close Bloomberg Connection

```
close(c)
```

See Also

Objects
`bloombergBPIPE`

Functions
`close` | `history` | `isconnection`

Related Examples

- “Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface” on page 5-28
- “Retrieve Bloomberg Intraday Tick Data Using Bloomberg B-PIPE C++ Interface” on page 5-35
- “Retrieve Bloomberg Real-Time Data Using Bloomberg B-PIPE C++ Interface” on page 5-37

Retrieve Bloomberg Intraday Tick Data Using Bloomberg B-PIPE C++ Interface

This example shows how to retrieve intraday tick data from Bloomberg.

Connect to Bloomberg

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Retrieve Intraday Tick Data

Retrieve the trade tick series for the past 50 days for the IBM security aggregated into 5-minute intervals.

```
d = timeseries(c,'IBM US Equity',{floor(now)-50,floor(now)},5,'Trade')
```

```
ans =

  Columns 1 through 7

   735487.40        187.20        187.60        187.02        187.08     207683.00        560.00
   735487.40        187.03        187.13        186.65        186.78      46990.00        349.00
   735487.40        186.78        186.78        186.40        186.47      51589.00        399.00
     ...

Column 8

   38902968.00
    8779374.00
    9626896.00
   ...
```

The columns in `d` contain the following:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row of data shows prices and tick data for the current date. The next row shows tick data for 5 minutes later.

Close Bloomberg Connection

```
close(c)
```

See Also

Objects
`bloombergBPIPE`

Functions
`close` | `isconnection` | `timeseries`

Related Examples

- “Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface” on page 5-28
- “Retrieve Bloomberg Historical Data Using Bloomberg B-PIPE C++ Interface” on page 5-31
- “Retrieve Bloomberg Real-Time Data Using Bloomberg B-PIPE C++ Interface” on page 5-37

Retrieve Bloomberg Real-Time Data Using Bloomberg B-PIPE C ++ Interface

This example shows how to retrieve real-time data from Bloomberg. Here, to display Bloomberg stock tick data at the command line, use the event handler `disp`. Instead of the default event handler, you can create your own event handler function to process Bloomberg data.

Connect to Bloomberg

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Retrieve Real-Time Data

Retrieve the last price and volume for IBM and Ford Motor Company securities.

```
[~,t] = realtime(c,{'IBM US Equity','F US Equity'}, ...
    {'LAST_PRICE','VOLUME'},'disp')
```

```
t =


   Timer Object: timer-4

   Timer Settings
      ExecutionMode: fixedRate
              Period: 0.05
           BusyMode: drop
            Running: off
```

```
   Callbacks
           TimerFcn: 1x5 cell array
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''

Columns 1 through 6

    {'SecurityID' }    {'LAST_PRICE'}    {'SecurityID' }    {'VOLUME'  }    {'SecurityID'   }
    {'F US Equity'}    {'8.960000'  }    {'F US Equity'}    {'13423731'}    {'IBM US Equity'}

  Columns 7 through 8

    {'SecurityID'   }    {'VOLUME'}
    {'IBM US Equity'}    {'744066'}
...
```

`realtime` returns the MATLAB timer object with its properties. Then, `realtime` returns the stock tick data for the IBM and Ford Motor Company securities with the last price and volume.

Stop the display of real-time data.

```
stop(t)
c.Session.stopSubscriptions
```

Close Bloomberg Connection

```
close(c)
```

See Also

Objects
`bloombergBPIPE`

Functions
`close` | `isconnection` | `realtime`

Related Examples

- “Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface” on page 5-28
- “Retrieve Bloomberg Historical Data Using Bloomberg B-PIPE C++ Interface” on page 5-31
- “Retrieve Bloomberg Intraday Tick Data Using Bloomberg B-PIPE C++ Interface” on page 5-35

More About

- “Writing and Running Custom Event Handler Functions” on page 1-26

Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface

This example shows how to retrieve current data from Bloomberg for a single security and for multiple securities.

Connect to Bloomberg

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Retrieve Current Data for Single Security

Retrieve last and open prices for Microsoft.

`d` contains the Bloomberg last and open prices as fields in a structure. `sec` contains the Bloomberg security name for Microsoft in a cell array. The security name is a character vector.

```
sec = 'MSFT US Equity';
fields = {'LAST_PRICE';'OPEN'}; % Retrieve data for last and open prices

[d,sec] = getdata(c,sec,fields)
```

```
d =

  struct with fields:

    LAST_PRICE: 62.30
          OPEN: 62.95


sec =

  cell

    'MSFT US Equity'
```

Retrieve Current Data for Multiple Securities

Retrieve last and open prices for the IBM and Ford Motor Company securities.

`d` contains the Bloomberg last and open prices as fields in a structure. `sec` contains the Bloomberg security names for IBM and Ford Motor Company in a cell array. Each security name is a character vector.

```
s = {'IBM US Equity','F US Equity'};
fields = {'LAST_PRICE';'OPEN'}; % Retrieve data for last and open prices

[d,sec] = getdata(c,s,fields)

d =

  struct with fields:

    LAST_PRICE: [2×1 double]
          OPEN: [2×1 double]


sec =

  2×1 cell array

    'IBM US Equity'
    'F US Equity'
```

Display the last price for both securities.

```
d.LAST_PRICE

ans =

        166.73
         12.63
```

Close Bloomberg Connection

```
close(c)
```

See Also

Objects
`bloombergServer`

Functions
`close` | `getdata` | `isconnection`

Related Examples

- “Retrieve Bloomberg Historical Data Using Bloomberg Server C++ Interface” on page 5-41
- “Retrieve Bloomberg Intraday Tick Data Using Bloomberg Server C++ Interface” on page 5-45
- “Retrieve Bloomberg Real-Time Data Using Bloomberg Server C++ Interface” on page 5-47

Retrieve Bloomberg Historical Data Using Bloomberg Server C ++ Interface

This example shows how to retrieve historical data from Bloomberg for a single security. The example shows retrieving weekly data within a date range and retrieving data with a default period. Then, the example also shows how to retrieve data for multiple securities.

Connect to Bloomberg

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Retrieve Historical Data for One Security

Retrieve monthly closing and open price data from January 1, 2012, through December 31, 2012, for Microsoft.

```
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
fromdate = '1/01/2012';
todate = '12/31/2012';
period = 'monthly';

[d,sec] = history(c,s,f,fromdate,todate,period)
```

```
d =

     734899.00         27.87         25.06
     734928.00         30.16         28.12
     734959.00         30.65         30.34
     ...

sec =

  cell

    'MSFT US Equity'
```

d contains the numeric representation of the date in the first column, closing price in the second column, and open price in the third column. Each row represents data for one month in the date range. sec contains the Bloomberg security name for Microsoft.

Retrieve Weekly Historical Data

Retrieve the weekly closing prices from November 1, 2010 through December 23, 2010 for the Microsoft security using US currency. In this case, the anchor date depends on the date December 23, 2010. Because this date is a Thursday, each previous value is reported for the Thursday of the week in question.

```
f = 'LAST_PRICE';
fromdate = '11/01/2010';
todate =  '12/23/2010';
period = {'weekly'};
currency = 'USD';

[d,sec] = history(c,s,f,fromdate,todate, ...
    period,currency)
```

```
d =

     734446.00         27.14
     734453.00         26.68
     734460.00         25.84
     734467.00         25.37
     734474.00         26.89
     734481.00         27.08
     734488.00         27.99
     734495.00         28.30


sec =

  1×1 cell array

    {'MSFT US Equity'}
```

d contains the numeric representation for the date in the first column and the closing price in the second column. sec contains the name of the Microsoft security.

Retrieve Historical Data Using Default Period

Retrieve the closing prices from August 1, 2010, through September 10, 2010, for the Microsoft security in US currency, and set the default period of the data by using []. The default period of a security depends on the security itself.

```
fromdate = '8/01/2010';
todate = '9/10/2010';
currency = 'USD';

[d,sec] = history(c,s,f,fromdate,todate, ...
    [],currency)
```

```
d =

     734352.00         26.33
     734353.00         26.16
```

```
     734354.00         25.73
     ...

sec =

  1×1 cell array

    {'MSFT US Equity'}
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the Microsoft security.

Retrieve Historical Data for Multiple Securities

Retrieve monthly closing and open prices from January 1, 2012, through December 31, 2012, for the IBM and Ford Motor Company securities.

`d` is a cell array of double matrices that contains the historical data for both securities. `sec` contains the Bloomberg security names for the IBM and Ford Motor Company securities in a cell array. Each security name is a character vector.

```
s = {'IBM US Equity','F US Equity'};
f = {'LAST_PRICE';'OPEN'};
fromdate = '1/01/2012';
todate = '12/31/2012';
period = 'monthly';

[d,sec] = history(c,s,f,fromdate,todate,period)

d =

  2×1 cell array

    [12×3 double]
    [12×3 double]


sec =

  2×1 cell array

    'IBM US Equity'
    'F US Equity'
```

Display the closing and open prices for the first security.

```
d{1}

ans =

     734899.00        192.60        186.73
     734928.00        196.73        193.21
     734959.00        208.65        197.23
     ...
```

The data in the double matrix is:

- First column — Numeric representation of the date
- Second column — Closing price

- Third column — Open price

Each row represents data for one month in the date range.

Close Bloomberg Connection

```
close(c)
```

See Also

Objects
`bloombergServer`

Functions
`close` | `history` | `isconnection`

Related Examples

- “Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface” on page 5-39
- “Retrieve Bloomberg Intraday Tick Data Using Bloomberg Server C++ Interface” on page 5-45
- “Retrieve Bloomberg Real-Time Data Using Bloomberg Server C++ Interface” on page 5-47

Retrieve Bloomberg Intraday Tick Data Using Bloomberg Server C++ Interface

This example shows how to retrieve intraday tick data from Bloomberg.

Connect to Bloomberg

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Retrieve Intraday Tick Data

Retrieve the trade tick series for the past 50 days for the IBM security aggregated into 5-minute intervals.

```
d = timeseries(c,'IBM US Equity',{floor(now)-50,floor(now)},5,'Trade')
```

```
ans =

  Columns 1 through 7

   735487.40        187.20        187.60        187.02        187.08     207683.00        560.00
   735487.40        187.03        187.13        186.65        186.78      46990.00        349.00
   735487.40        186.78        186.78        186.40        186.47      51589.00        399.00
     ...

Column 8

   38902968.00
    8779374.00
    9626896.00
   ...
```

The columns in `d` contain the following:

- Numeric representation of date and time
- Open price
- High price
- Low price

- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row of data shows prices and tick data for the current date. The next row shows tick data for 5 minutes later.

Close Bloomberg Connection

```
close(c)
```

See Also

Objects
`bloombergServer`

Functions
`close` | `isconnection` | `timeseries`

Related Examples

- “Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface” on page 5-39
- “Retrieve Bloomberg Historical Data Using Bloomberg Server C++ Interface” on page 5-41
- “Retrieve Bloomberg Real-Time Data Using Bloomberg Server C++ Interface” on page 5-47

Retrieve Bloomberg Real-Time Data Using Bloomberg Server C ++ Interface

This example shows how to retrieve real-time data from Bloomberg. Here, to display Bloomberg stock tick data at the command line, use the event handler `disp`. Instead of the default event handler, you can create your own event handler function to process Bloomberg data.

Connect to Bloomberg

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Retrieve Real-Time Data

Retrieve the last price and volume for IBM and Ford Motor Company securities.

```
[~,t] = realtime(c,{'IBM US Equity','F US Equity'}, ...
    {'LAST_PRICE','VOLUME'},'disp')
```

```
t =


   Timer Object: timer-4

   Timer Settings
      ExecutionMode: fixedRate
              Period: 0.05
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: 1x5 cell array
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''

Columns 1 through 6
```

```
    {'SecurityID' }    {'LAST_PRICE'}    {'SecurityID' }    {'VOLUME'  }    {'SecurityID'   }
    {'F US Equity'}    {'8.960000'  }    {'F US Equity'}    {'13423731'}    {'IBM US Equity'}

  Columns 7 through 8

    {'SecurityID'   }    {'VOLUME'}
    {'IBM US Equity'}    {'744066'}
...
```

`realtime` returns the MATLAB timer object with its properties. Then, `realtime` returns the stock tick data for the IBM and Ford Motor Company securities with the last price and volume.

Stop the display of real-time data.

```
stop(t)
c.Session.stopSubscriptions
```

Close Bloomberg Connection

```
close(c)
```

See Also

Objects
`bloombergServer`

Functions
`close` | `isconnection` | `realtime`

Related Examples

- “Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface” on page 5-39
- “Retrieve Bloomberg Historical Data Using Bloomberg Server C++ Interface” on page 5-41
- “Retrieve Bloomberg Intraday Tick Data Using Bloomberg Server C++ Interface” on page 5-45

More About

- “Writing and Running Custom Event Handler Functions” on page 1-26

6

Transaction Cost Analysis

Analyze Trading Execution Results

This example shows how to conduct post-trade analysis using transaction cost analysis from the Kissell Research Group. Post-trade analysis includes implementation shortfall, alpha capture, benchmark costs, broker value add, and Z-Score. For details about these metrics, see “Post-Trade Analysis Metrics Definitions” on page 6-5. You can use post-trade analysis to evaluate portfolio returns and profits. You can measure performance of brokers and algorithms.

To access the example code, enter `edit KRGPostTradeAnalysisExample.m` at the command line.

Retrieve Market-Impact Parameters and Load Transaction Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`.

```
k = krg(miData);
```

Load the example data `PostTradeData` from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox.

```
load KRGExampleData.mat PostTradeData
```

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Determine Implementation Shortfall Costs

Determine the components of the implementation shortfall costs in basis points. The components are:

- Fixed cost `ISFixed`
- Delay cost `ISDelayCost`
- Execution cost `ISExecutionCost`
- Opportunity cost `ISOpportunityCost`

For details about the cost components, see “Post-Trade Analysis Metrics Definitions” on page 6-5.

```
PostTradeData.ISDollars = ...
    PostTradeData.OrderShares .* PostTradeData.ISDecisionPrice;
PostTradeData.ISFixed = ...
    PostTradeData.ISFixedDollars ./ PostTradeData.ISDollars*10000;
PostTradeData.ISDelayCost = ...
    PostTradeData.OrderShares .* ...
    (PostTradeData.ISArrivalPrice-PostTradeData.ISDecisionPrice).* ...
    PostTradeData.SideIndicator ./ PostTradeData.ISDollars*1000;
PostTradeData.ISExecutionCost = ...
```

```
    PostTradeData.TradedShares .* ...
    (PostTradeData.AvgExecPrice-PostTradeData.ISArrivalPrice).* ...
    PostTradeData.SideIndicator ./ PostTradeData.ISDollars*1000;
PostTradeData.ISOpportunityCost = ...
    (PostTradeData.OrderShares-PostTradeData.TradedShares).* ...
    (PostTradeData.ISEndPrice-PostTradeData.ISArrivalPrice).* ...
    PostTradeData.SideIndicator ./ PostTradeData.ISDollars*1000;
```

Determine the total implementation shortfall cost `ISCost`.

```
PostTradeData.ISCost = PostTradeData.ISFixed + ...
    PostTradeData.ISDelayCost + PostTradeData.ISExecutionCost + ...
    PostTradeData.ISOpportunityCost;
```

Determine Profit

Determine the alpha capture `Alpha_CapturePct`. Divide realized profit `Alpha_Realized` by potential profit `Alpha_TotalPeriod`.

```
PostTradeData.Alpha_Realized = ...
    (PostTradeData.ISEndPrice-PostTradeData.AvgExecPrice).* ...
    PostTradeData.TradedShares .* PostTradeData.SideIndicator ./ ...
    (PostTradeData.TradedShares .* PostTradeData.ISArrivalPrice)*10000;
PostTradeData.Alpha_TotalPeriod = ...
    (PostTradeData.ISEndPrice-PostTradeData.ISArrivalPrice).* ...
    PostTradeData.TradedShares .* PostTradeData.SideIndicator ./ ...
    (PostTradeData.TradedShares .* PostTradeData.ISArrivalPrice)*10000;
lenAlpha_Realized = length(PostTradeData.Alpha_Realized);
PostTradeData.Alpha_CapturePct = zeros(lenAlpha_Realized,1);
for ii = 1:lenAlpha_Realized
    if PostTradeData.Alpha_TotalPeriod(ii) > 0
        PostTradeData.Alpha_CapturePct(ii) = ...
        PostTradeData.Alpha_Realized(ii) ./ ...
        PostTradeData.Alpha_TotalPeriod(ii);
    else
        PostTradeData.Alpha_CapturePct(ii) = ...
        -(PostTradeData.Alpha_Realized(ii) - ...
        PostTradeData.Alpha_TotalPeriod(ii)) ./ ...
        PostTradeData.Alpha_TotalPeriod(ii);
    end
end
```

Determine Benchmark and Trading Costs

Determine benchmark costs in basis points. Here, the benchmark prices are:

- Close price of the previous day `PrevClose_Cost`
- Open price `Open_Cost`
- Close price `Close_Cost`
- Arrival cost `Arrival_Cost`
- Period VWAP `PeriodVWAP_Cost`

```
PostTradeData.PrevClose_Cost = ...
    (PostTradeData.AvgExecPrice-PostTradeData.PrevClose).* ...
    PostTradeData.SideIndicator ./ PostTradeData.PrevClose*10000;
PostTradeData.Open_Cost = ...
    (PostTradeData.AvgExecPrice-PostTradeData.Open).* ...
    PostTradeData.SideIndicator ./ PostTradeData.Open*10000;
```

```
PostTradeData.Close_Cost = (PostTradeData.AvgExecPrice-PostTradeData.Close).* ...
    PostTradeData.SideIndicator ./ PostTradeData.Close*10000;
PostTradeData.Arrival_Cost = (PostTradeData.AvgExecPrice- ...
    PostTradeData.ArrivalPrice).* ...
    PostTradeData.SideIndicator ./ PostTradeData.ArrivalPrice*10000;
PostTradeData.PeriodVWAP_Cost = (PostTradeData.AvgExecPrice- ...
    PostTradeData.PeriodVWAP).* ...
    PostTradeData.SideIndicator ./ PostTradeData.PeriodVWAP*10000;
```

Estimate market-impact `miCost` and timing risk `tr` costs.

```
PostTradeData.Size = PostTradeData.TradedShares ./ PostTradeData.ADV;
PostTradeData.Price = PostTradeData.ArrivalPrice;
PostTradeData.miCost = marketImpact(k,PostTradeData);
PostTradeData.tr = timingRisk(k,PostTradeData);
```

Determine Broker Value Add and Z-Score

Determine the broker value add using the arrival cost and market impact.

```
PostTradeData.ValueAdd = (PostTradeData.Arrival_Cost-PostTradeData.miCost) * -1;
```

Determine the Z-Score using the broker value add and timing risk.

```
PostTradeData.zScore = PostTradeData.ValueAdd./PostTradeData.tr;
```

For details about the preceding calculations, contact the Kissell Research Group.

See Also

`krg` | `marketImpact` | `timingRisk`

More About

- “Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17
- “Post-Trade Analysis Metrics Definitions” on page 6-5
- “Kissell Research Group Data Sets” on page 6-7

Post-Trade Analysis Metrics Definitions

After executing a transaction, Kissell Research Group provides various metrics for analyzing the results of a transaction. For an example using these metrics, see “Analyze Trading Execution Results” on page 6-2.

For details about these calculations, contact the Kissell Research Group.

Implementation Shortfall

Implementation shortfall (IS) determines the total cost of implementing an investment decision. IS subtracts the actual return from the paper return of a stock or portfolio after including all visible costs including commissions, fees, and taxes. The Kissell Research Group IS cost formula decomposes costs into fixed, delay, execution, and opportunity cost components.

IS Component	Description
Fixed cost	Cost component that is not dependent upon the implementation strategy.
Delay cost	Cost component that represents the loss in investment value between the time the managers make the investment decision and the order releases to the market.
Execution cost	Cost component that is the difference between the execution price and the stock price at the time the order releases to the market.
Opportunity cost	Cost component that represents the foregone profit or loss resulting from not being able to execute the order to completion within the allotted time period.

Portfolio managers and traders use IS to understand the trading cost environment.

Alpha Capture

Alpha capture, or profit, is the realized profit divided by the potential profit. Realized profit is based on the difference between end price and average execution price. Potential profit is based on the difference between end price and arrival price. Portfolio managers and traders use alpha capture to measure portfolio performance.

Benchmark Costs

The benchmark cost compares the average execution price to a specific benchmark price. A benchmark price can be any price such as the close price. Traders use benchmark costs to measure strategy and transaction performance.

Broker Value Add

The broker value add metric is a measure of the overall broker performance. A positive value indicates that the broker performed better than expected and a negative value indicates the broker

under-performed expectations. This metric is the difference between the estimated trading cost and the actual cost incurred by the investor. You can estimate trading costs using `marketImpact`, `priceAppreciation`, and `timingRisk`. This metric reflects performance given all market conditions on the day and buying and selling behavior from all other participants.

Traders use this metric to measure broker performance.

Z-Score

Z-Score is the broker value add metric divided by timing risk. You can estimate timing risk using `timingRisk`. The Z-Score specifies the number of standard deviations away from the estimated cost. If the Z-Score is greater than or equal to two standard deviations, then the actual cost varies greatly from the estimated cost.

Traders use this metric to measure broker performance.

References

[1] Kissell, Robert. “The Expanded Implementation Shortfall: Understanding Transaction Cost Components.” *Journal of Trading*. Vol. 1, Number 3, Summer 2006, pp. 6–16.

See Also

Related Examples

- “Analyze Trading Execution Results” on page 6-2

Kissell Research Group Data Sets

The following descriptions define the data sets provided in the file `KRGExampleData.mat`.

Basket Variables

The table `Basket` contains a trade list for a collection of stocks in a portfolio. For examples of using this data set, see “Rank Broker Performance” on page 6-55.

Real trade lists come from portfolio managers.

Table Variable	Description
`Symbols`	Stock symbol.
`Side`	Side (`'B'` or `'S'`).
`Size`	Size (number of shares divided by average daily volume).
`Shares`	Number of shares.
`Price`	Stock price.
`ADV`	Average daily volume.
`Volatility`	A statistical measure of the dispersion of daily returns for a given security. Volatility is the standard deviation of daily log price returns over time. Kissell Research Group uses a 30-day historical period. Annualize volatility by multiplying by the square root of 250.
`POV`	Percentage of volume.

BrokerNames Variables

The table `BrokerNames` contains the broker names and their associated market-impact code. For examples of using this data set, see “Rank Broker Performance” on page 6-55.

Real trade lists come from portfolio managers.

Table Variable	Description
`Broker`	Broker name.
`MICode`	Market-impact code (1, 2, 3, and so on).

TradeData Variables

The table `TradeData` provides example data for a collection of stocks in a transaction. For examples of using this data set, see “Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17 and “Estimate Portfolio Liquidation Costs” on page 6-20.

Real market data comes from a data source such as Bloomberg.

Table Variable	Description
`Symbol`	Stock symbol.
`Side`	Side (`'Buy'` or `'Sell'`).
`SideIndicator`	Side indicator. `1` is a buy (add shares to portfolio). `-1` is a sell (remove shares from portfolio).
`AvgExecPrice`	Average execution price.
`ArrivalPrice`	Arrival price. The price at the time the order enters the market.
`PeriodVWAP`	Volume weighted average price (VWAP). The VWAP compares the execution price to the interval VWAP price.
`CCYRate`	Currency rate.
`Volatility`	A statistical measure of the dispersion of daily returns for a given security. Volatility is the standard deviation of daily log price returns over time. Kissell Research Group uses a 30-day historical period. Annualize volatility by multiplying by the square root of 250.
`POV`	Percentage of volume.
`SectorCategory`	Market sector category (`'Energy'`, `'Industrials'`, `'Materials'`, and so on).
`OrderSizeCategory`	Order size category (`'Large'`, `'Medium'`, or `'Small'`).
`VolatilityCategory`	Volatility category (`'High'`, `'Medium'`, or `'Low'`).
`POVRateCategory`	Percentage of volume rate category (`'Aggressive'`, `'Passive'`, or `'Normal'`).
`MktCapCategory`	Market capitalization category (`'LC'` is large cap, `'MC'` is mid cap, `'SM'` is small cap).
`MomentumCategory`	Momentum category (`'Favorable'`, `'Neutral'`, or `'Adverse'`).
`MktMovementCategory`	Market movement category (`'Favorable'`, `'Neutral'`, or `'Adverse'`).
`ADV`	Average daily volume.
`Price`	Stock price.
`Size`	Size (number of shares divided by average daily volume).
`Alpha_bp`	Alpha estimate per day in basis points.
`Shares`	Number of shares.
`Broker`	Broker name.

TradeDataCurrent and TradeDataHistorical Variables

The tables `TradeDataCurrent` and `TradeDataHistorical` provide example current and historical data, respectively, for a collection of stocks in a transaction. For an example of using this data set, see “Determine Buy-Sell Imbalance Using Cost Index” on page 6-51.

Real market data comes from a data source such as Bloomberg.

Table Variable	Description
`Symbol`	Stock symbol.
`Date`	Transaction date.
`MICode`	Market-impact code (1, 2, 3, and so on).
`Open`	Stock open price.
`VWAP`	Volume weighted average price (VWAP).
`Last`	Stock last price.
`Volume`	Trade volume.
`Volatility`	A statistical measure of the dispersion of daily returns for a given security. Volatility is the standard deviation of daily log price returns over time. Kissell Research Group uses a 30-day historical period. Annualize volatility by multiplying by the square root of 250.
`ADV`	Average daily volume.
`Beta`	Beta.
`IndexOpen`	Index open price.
`IndexVWAP`	Index VWAP.
`IndexLast`	Index last price.
`Price`	Stock price.
`POV`	Percentage of volume.
`Shares`	Number of shares.

PortfolioData Variables

The table `PortfolioData` provides example data for a collection of stocks in a portfolio. To use this data set, see `portfolioCostCurves`.

Real portfolio data comes from a portfolio belonging to a company or portfolio manager.

Table Variable	Description
`Symbol`	Stock symbol.
`Price_Local`	Local price of the stock.

Table Variable	Description
`Price_Currency`	Stock price with a specified base currency if the stock trades outside the United States. If the stock trades in the United States, `Price_Currency` has the same value as `Price_Local`.
`ADV`	Average daily volume.
`Volatility`	Volatility.
`Shares`	Number of shares.

PostTradeData Variables

The table `PostTradeData` provides example data for a collection of stocks in an executed transaction. To use this data set, see “Analyze Trading Execution Results” on page 6-2.

Real market data comes from a data source such as Bloomberg.

Table Variable	Description
`Symbol`	Stock symbol.
`Side`	Side (`'Buy'` or `'Sell'`).
`SideIndicator`	Side indicator. `1` is a buy (add shares to portfolio). `-1` is a sell (remove shares from portfolio).
`Date`	Transaction date.
`DecisionTime`	Decision time. The portfolio manager decides to buy, sell, short, or cover a position at this time. If no other timestamp is available, set this variable to the time when the portfolio manager enters the order into the trading system. If the portfolio manager does not have a timestamp for this decision, investors use the close time of the previous day, open time, or arrival time.
`ArrivalTime`	Arrival time. The trading system enters the order into the market for execution at this time. You can obtain it from the first trade from the electronic audit trail.
`EndTime`	End time. The portfolio manager specifies to complete the order at this time. Typically, this time is the end of the day or the time of the last trade.
`AvgExecPrice`	Average executed price.
`OrderShares`	Number of shares.
`TradedShares`	Number of shares executed.
`Volatility`	Volatility.
`ADV`	Average daily volume.
`POV`	Percentage of volume.

Table Variable	Description
`CCYRate`	Currency rate.
`MICategory`	Market-impact category (for example, `1`).
`PrevClose`	Close price of the previous day.
`Open`	Open price.
`Close`	Close price.
`ArrivalPrice`	Arrival price. The price at the time the order enters the market.
`PeriodVWAP`	Volume weighted average price (VWAP). The VWAP compares the execution price to the interval VWAP price.
`Broker`	Broker name.
`Algorithm`	Trading algorithm (`'Dark Pool'`, `'TWAP'`, `'Arrival'`, and so on).
`Manager`	Portfolio manager name.
`Trader`	Trader name.
`SectorCategory`	Market sector category (`'Energy'`, `'Industrials'`, `'Materials'`, and so on).
`OrderSizeCategory`	Order size category (`'Large'`, `'Medium'`, or `'Small'`).
`VolatilityCategory`	Volatility category (`'High'`, `'Medium'`, or `'Low'`).
`POVRateCategory`	Percentage of volume rate category (`'Aggressive'`, `'Passive'`, or `'Normal'`).
`MktCapCategory`	Market capitalization category (`'LC'` is large cap, `'MC'` is mid cap, `'SM'` is small cap).
`StockMomentumCategory`	Stock momentum category (`'Favorable'`, `'Neutral'`, or `'Adverse'`).
`MktMovementCategory`	Market movement category (`'Favorable'`, `'Neutral'`, or `'Adverse'`).
`StepOut`	Investor field designation. This variable is optional for grouping and summary analysis. This field refers to a process where a broker (broker 1) receives an order from a client. Then this broker gives that order to another broker (broker 2) for its execution. Broker 1 receives credit for the trade but its performance applies to broker 2 who executed the trade.
`ISDecisionPrice`	Decision price. This variable is the stock price when the portfolio manager decides to buy, sell, short, or cover a position.
`ISArrivalPrice`	Midpoint of the bid-ask spread at the time an order enters the market.

Table Variable	Description
ISEndPrice	End price. This variable is the stock price at the specified end time of the order.
ISFixedDollars	Fixed fees in dollars that include the commission, taxes, clearing and settlement charges, and so on.

TradeDataBackTest Variables

The table TradeDataBackTest provides example data for a set of stocks and a series of dates. The data contains historical trade information for each stock. To use this data set, see “Conduct Back Test on Portfolio” on page 6-35.

Real market data comes from a data source such as Bloomberg.

Table Variable	Description
Symbol	Stock symbol.
Date	Historical transaction date.
Shares	Number of shares.
Side	Side ('Buy' or 'Sell').
Value	Dollar value of the stock in the portfolio.
Price	Stock price.
Size	Size (number of shares divided by average daily volume).
EstReturn	Estimated return decimal value for the stock in the portfolio.
Volatility	A statistical measure of the dispersion of daily returns for a given security. Volatility is the standard deviation of daily log price returns over time. Kissell Research Group uses a 30-day historical period. Annualize volatility by multiplying by the square root of 250.
ADV	Average daily volume.
MktCap	Market capitalization.
TradeTime	Trade duration time.
POVRate	Percentage of volume rate.
MICode	Market-impact code (1, 2, 3, and so on).
FXRate	Foreign exchange rate.
POV	Percentage of volume.

TradeDataStressTest Variables

The table TradeDataStressTest provides example data for a set of stocks for a date range. The data contains trade information for each stock. To use this data set, see “Conduct Stress Test on Portfolio” on page 6-38.

Real market data comes from a data source such as Bloomberg.

Table Variable	Description
`Symbol`	Stock symbol.
`Date`	Historical transaction date.
`Shares`	Number of shares.
`Side`	Side (`'Buy'` or `'Sell'`).
`Value`	Dollar value of the stock in the portfolio.
`Price`	Stock price.
`Size`	Size (number of shares divided by average daily volume).
`EstReturn`	Estimated return decimal value for the stock in the portfolio.
`Volatility`	A statistical measure of the dispersion of daily returns for a given security. Volatility is the standard deviation of daily log price returns over time. Kissell Research Group uses a 30-day historical period. Annualize volatility by multiplying by the square root of 250.
`ADV`	Average daily volume.
`MktCap`	Market capitalization.
`TradeTime`	Trade duration time.
`POVRate`	Percentage of volume rate.
`MICode`	Market-impact code (1, 2, 3, and so on).
`FXRate`	Foreign exchange rate.

TradeDataPortOpt Variables

The table `TradeDataPortOpt` contains example data for a collection of stocks in a portfolio. This data contains lower and upper bounds for the constraints used in the portfolio optimization. To use this data set, see “Liquidate Dollar Value from Portfolio” on page 6-43.

To see the related covariance data for each stock in the portfolio, see the covariance data table `CovarianceData`.

Real portfolio data comes from a portfolio belonging to a company or portfolio manager.

Table Variable	Description
`Symbol`	Stock symbol.
`Date`	Transaction date.
`Shares`	Number of shares.
`Value`	Dollar value of the stock in the portfolio.
`Price`	Stock price.

Table Variable	Description
`Size`	Size (number of shares divided by average daily volume).
`EstReturn`	Estimated return decimal value for the stock in the portfolio.
`Volatility`	A statistical measure of the dispersion of daily returns for a given security. Volatility is the standard deviation of daily log price returns over time. Kissell Research Group uses a 30-day historical period. Annualize volatility by multiplying by the square root of 250.
`ADV`	Average daily volume.
`MktCap`	Market capitalization.
`TradeTime`	Trade time.
`MICode`	Market-impact code (1, 2, 3, and so on).
`LB_Wt`	Lower bound weight.
`UB_Wt`	Upper bound weight.
`LB_MinShares`	Lower bound for the minimum shares.
`UB_MaxShares`	Upper bound for the maximum shares.
`LB_MinPctADV`	Lower bound for the minimum percentage of average daily volume.
`UB_MaxPctADV`	Upper bound for the maximum percentage of average daily volume.
`LB_MinValue`	Lower bound for the minimum value.
`UB_MaxValue`	Upper bound for the maximum value.
`UB_MaxMI`	Upper bound for the maximum market-impact cost.

TradeDataTradeOpt Variables

The table `TradeDataTradeOpt` provides an example trade list for a collection of stocks in a portfolio. For an example of using this data set, see “Optimize Trade Schedule Trading Strategy for Basket” on page 6-60.

Real trade lists come from portfolio managers.

Table Variable	Description
`Date`	Transaction date.
`Side`	Side (`'B'` or `'S'`).
`Shares`	Number of shares.
`Price`	Stock price.
`ADV`	Average daily volume.

Table Variable	Description
`Volatility`	A statistical measure of the dispersion of daily returns for a given security. Volatility is the standard deviation of daily log price returns over time. Kissell Research Group uses a 30-day historical period. Annualize volatility by multiplying by the square root of 250.
`PctADV`	Percentage of average daily volume.
`Value`	Transaction value.
`Weight`	Weight.
`SideIndicator`	Side indicator. `1` is a buy (add shares to portfolio). `-1` is a sell (remove shares from portfolio).
`MIRegion`	Market-impact region.
`Symbol`	Stock symbol.
`Alpha_bp`	Alpha in basis points.
`Beta`	Beta.
`Sector`	Market sector, such as Energy.
`MktCap`	Market capitalization.

CovarianceData Table

The table `CovarianceData` contains a covariance value for all stocks in the portfolio data table `TradeDataPortOpt`. Each variable in the table is a different stock. To use this data set in the portfolio optimization, see “Liquidate Dollar Value from Portfolio” on page 6-43.

CovarianceTradeOpt Table

The table `CovarianceTradeOpt` contains a covariance value for each stock in the portfolio data table `TradeDataTradeOpt`. Each variable in the table is a different stock. To use this data set in the trade schedule optimization, see “Optimize Trade Schedule Trading Strategy for Basket” on page 6-60.

References

[1] Kissell, Robert. “A Practical Framework for Transaction Cost Analysis.” *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. “The Expanded Implementation Shortfall: Understanding Transaction Cost Components.” *Journal of Trading*. Vol. 1, Number 3, Summer 2006, pp. 6–16.

[3] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

Related Examples

- “Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17
- “Conduct Back Test on Portfolio” on page 6-35
- “Conduct Stress Test on Portfolio” on page 6-38
- “Estimate Portfolio Liquidation Costs” on page 6-20
- “Liquidate Dollar Value from Portfolio” on page 6-43
- “Analyze Trading Execution Results” on page 6-2

Conduct Sensitivity Analysis to Estimate Trading Costs

This example shows how to evaluate changes in trading costs due to liquidity, volatility, and market sensitivity to order flow and trades. With transaction cost analysis from the Kissell Research Group, you can simulate the trading cost environment for a collection of stocks. Sensitivity analysis enables you to estimate future trading costs for different market conditions to determine the appropriate portfolio contents that meet the needs of the investors.

Here, evaluate changes in trading costs due to decreasing average daily volume by 50% and doubling volatility. The example data uses the percentage of volume (POV) trade strategy.

To access the example code, enter `edit KRGSensitivityAnalysisExample.m` at the command line.

Retrieve Market-Impact Parameters and Load Transaction Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`.

```
k = krg(miData);
```

Load the example data from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox.

```
load KRGExampleData.mat
```

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Estimate Initial Trading Costs

Estimate initial trading costs using the example data `TradeData`. The trading costs are:

- Instantaneous trading cost `itc`
- Market-impact cost `mi`
- Timing risk `tr`
- Price appreciation `pa`

Group all four trading costs into a numeric matrix `initTCA`.

```
itc = iStar(k,TradeData);
mi = marketImpact(k,TradeData);
tr = timingRisk(k,TradeData);
pa = priceAppreciation(k,TradeData);
initTCA = [itc mi tr pa];
```

Create Scenario

Set variables to create the scenario. Here, the scenario decreases average daily volume by 50% and doubles volatility. The stock price, volume, estimated alpha, and trade strategy remain unchanged from the example data. You can modify the values of these variables to create different scenarios. The fields are:

- Average daily volume
- Volatility
- Stock price
- Volume
- Alpha estimate
- POV trade strategy
- Trade time trade strategy

```
adjADV = 0.5;
adjVolatility = 2.0;
adjPrice = 1.0;
adjVolume = 1.0;
adjAlpha = 1.0;
adjPOV = 1.0;
adjTradeTime = 1.0;
```

Adjust the example data based on the scenario variables.

```
TradeDataAdj = TradeData;
TradeDataAdj.Size = TradeData.Size .* (1./adjADV);
TradeDataAdj.ADV = TradeData.ADV .* adjADV;
TradeDataAdj.Volatility = TradeData.Volatility .* adjVolatility;
TradeDataAdj.Price = TradeData.Price .* adjPrice;
TradeDataAdj.Alpha_bp = TradeData.Alpha_bp .* adjAlpha;
```

`TradeDataAdj` contains the adjusted data. Size doubles because average daily volume decreases by 50%.

Convert POV trade strategy to the trade time trade strategy.

```
[~,povFlag,timeFlag] = krg.krgDataFlags(TradeData);
if povFlag
    TradeDataAdj.POV = TradeData.POV.*adjPOV;
    TradeDataAdj.TradeTime = TradeDataAdj.Size .* ...
        ((1-TradeDataAdj.POV) ./ TradeDataAdj.POV) .* (1./adjVolume);
elseif timeFlag
    TradeDataAdj.TradeTime = tradedata.TradeTime .* adjTradeTime;
    TradeDataAdj.POV = TradeDataAdj.Size ./ ...
        (TradeDataAdj.Size + TradeDataAdj.TradeTime .* adjVolume);
end
```

Estimate Trading Costs for Scenario

Estimate the trading costs based on the adjusted data. The numeric matrix `newTCA` contains the trading costs for the scenario.

```
itc = iStar(k,TradeDataAdj);
mi = marketImpact(k,TradeDataAdj);
```

```
tr = timingRisk(k,TradeDataAdj);
pa = priceAppreciation(k,TradeDataAdj);
newTCA = [itc mi tr pa];
```

Subtract the trading costs from the scenario from the initial trading costs.

```
rawWI = newTCA - initTCA;
wi = table(rawWI(:,1),rawWI(:,2),rawWI(:,3),rawWI(:,4), ...
    'VariableNames',{'ITC','MI','TR','PA'});
```

The table `wi` contains the full impact of this scenario on the trading costs.

Display trading costs for the first three rows in `wi`.

```
wi(1:3,:)
```

```
ans =

     ITC        MI        TR        PA
    _____    ______    ______    _____

     43.05      0.65    290.80    -9.49
    408.29    124.52    443.16     8.47
     80.92     13.79    114.97     0.93
```

The variables in `wi` are:

- Instantaneous trading cost
- Market-impact cost
- Timing risk
- Price appreciation

For details about the preceding calculations, contact the Kissell Research Group.

See Also

`krg` | `iStar` | `marketImpact` | `timingRisk` | `priceAppreciation`

More About

- “Analyze Trading Execution Results” on page 6-2
- “Kissell Research Group Data Sets” on page 6-7

Estimate Portfolio Liquidation Costs

This example shows how to determine the cost of liquidating individual stocks in a portfolio using transaction cost analysis from the Kissell Research Group. Compare the individual stocks in a portfolio using various metrics in a scatter plot.

The example data uses the percentage of volume trade strategy to calculate costs. You can also use the trade time trade strategy to run the analysis by replacing the percentage of volume data with trade time data.

To access the example code, enter `edit KRGPortfolioLiquidityExample.m` at the command line.

Retrieve Market-Impact Parameters and Load Transaction Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`.

```
k = krg(miData);
```

Load the example data `TradeData` from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox.

```
load KRGExampleData.mat TradeData
```

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Estimate Trading Costs

Estimate market-impact costs `mi`.

```
TradeData.mi = marketImpact(k,TradeData);
```

Estimate the timing risk `tr`.

```
TradeData.tr = timingRisk(k,TradeData);
```

Estimate the liquidity factor `lf`.

```
TradeData.lf = liquidityFactor(k,TradeData);
```

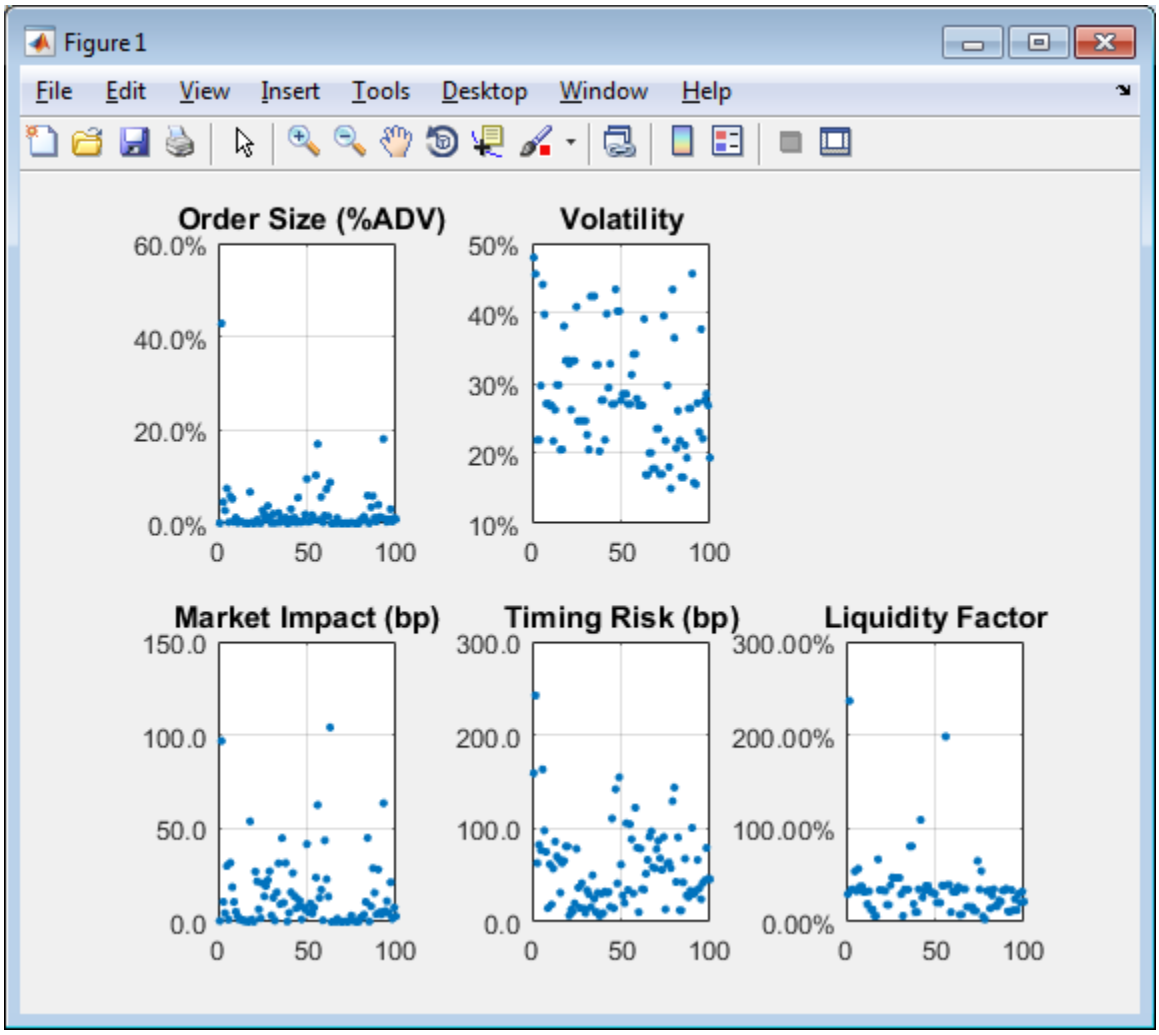
For details about the preceding calculations, contact the Kissell Research Group.

Display Portfolio Plots

Create a scatter plot that shows the following:

- Size
- Volatility
- Market impact
- Timing risk
- Liquidity factor

```
figure
axOrder = subplot(2,3,1);
nSymbols = 1:length(TradeData.Size);
scatter(nSymbols,TradeData.Size*100,10,'filled')
grid on
box on
title(' Order Size (%ADV)')
axOrder.YAxis.TickLabelFormat = '%.1f%%';

axVolatility = subplot(2,3,2);
scatter(nSymbols,TradeData.Volatility*100,10,'filled')
grid on
box on
title('Volatility')
axVolatility.YAxis.TickLabelFormat = '%g%%';

axMI = subplot(2,3,4);
scatter(nSymbols,TradeData.mi,10,'filled')
grid on
box on
title('Market Impact (bp)')
axMI.YAxis.TickLabelFormat = '%.1f';

axTR = subplot(2,3,5);
scatter(nSymbols,TradeData.tr,10,'filled')
grid on
box on
title('Timing Risk (bp)')
axTR.YAxis.TickLabelFormat = '%.1f';

axLF = subplot(2,3,6);
scatter(nSymbols,TradeData.lf*100,10,'filled')
grid on
box on
title('Liquidity Factor')
axLF.YAxis.TickLabelFormat = '%.2f%%';
```



This figure demonstrates a snapshot view into the trading and liquidation costs, volatility, and size of the stocks in the portfolio. You can modify this scatter plot to include other variables from `TradeData`.

See Also

`krg` | `marketImpact` | `timingRisk` | `liquidityFactor`

More About

- “Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17
- “Kissell Research Group Data Sets” on page 6-7

Optimize Percentage of Volume Trading Strategy

This example shows how to optimize the strategy for a single stock by minimizing trading costs using transaction cost analysis from the Kissell Research Group. The optimization minimizes trading costs associated with the percentage of volume trading strategy and a specified risk aversion parameter *Lambda*. The trading cost minimization is expressed as

$$\min[(MI + PA) + Lambda \cdot TR],$$

where trading costs are market impact *MI*, price appreciation *PA*, and timing risk *TR*. For details, see `marketImpact`, `priceAppreciation`, and `timingRisk`. This example finds a local minimum for this expression. For details about searching for the global minimum, see "Optimization Troubleshooting and Tips".

Here, you can optimize the percentage of volume trade strategy. To optimize trade time and trade schedule strategies, see "Optimize Trade Time Trading Strategy" on page 6-26 and "Optimize Trade Schedule Trading Strategy" on page 6-29.

To access the example code, enter `edit KRGSingleStockOptimizationExample.m` at the command line.

Retrieve Market-Impact Parameters and Create Example Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`.

```
k = krg(miData);
```

Create Single Stock Data

The structure `tradeData` contains data for a single stock. Use a structure or table to define this data. The fields are:

- Number of shares
- Average daily volume
- Volatility
- Stock price
- Initial percentage of volume trade strategy
- Alpha estimate

```
tradeData.Shares = 100000;
tradeData.ADV = 1000000;
tradeData.Volatility = 0.25;
```

```
tradeData.Price = 35;
tradeData.POV = 0.5;
tradeData.Alpha_bp = 50;
```

Define Optimization Parameters

Define risk aversion level `Lambda`. Set `Lambda` from 0 to `Inf`.

```
Lambda = 1;
```

Define lower `LB` and upper `UB` bounds of strategy input for optimization.

```
LB = 0;
UB = 1;
```

Define the function handle `fun` for the objective function. To access the code for this function, enter `edit krgSingleStockOptimizer.m`.

```
fun = @(pov)krgSingleStockOptimizer(pov,k,tradeData,Lambda);
```

Minimize Trading Costs for Trade Strategy

Minimize the trading costs for the percentage of volume trade strategy. `fminbnd` finds the optimal value for the percentage of volume trade strategy based on the lower and upper bound values. `fminbnd` finds a local minimum for the trading cost minimization expression.

```
[tradeData.POV,totalcost] = fminbnd(fun,LB,UB);
```

Display the optimized trade strategy `tradeData.POV`.

```
tradeData.POV
```

```
ans =

    0.35
```

Estimate Trading Costs for Optimized Strategy

Estimate the trading costs `povCosts` using the optimized trade strategy.

```
mi = marketImpact(k,tradeData);
pa = priceAppreciation(k,tradeData);
tr = timingRisk(k,tradeData);
povCosts = [totalcost mi pa tr];
```

Display trading costs.

```
povCosts
```

```
        100.04         56.15          4.63         39.27
```

The trading costs are:

- Total cost
- Market impact
- Price appreciation
- Timing risk

For details about the preceding calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. “Algorithmic Trading Strategies.” Ph.D. Thesis. Fordham University, May 2006.

[2] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[3] Glantz, Morton, and Robert Kissell. *Multi-Asset Risk Modeling*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`fminbnd` | `marketImpact` | `priceAppreciation` | `timingRisk` | `krg`

Related Examples

- “Optimize Trade Time Trading Strategy” on page 6-26
- “Optimize Trade Schedule Trading Strategy” on page 6-29
- “Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17
- “Estimate Portfolio Liquidation Costs” on page 6-20

Optimize Trade Time Trading Strategy

This example shows how to optimize the strategy for a single stock by minimizing trading costs using transaction cost analysis from the Kissell Research Group. The optimization minimizes trading costs associated with the trade time trading strategy and a specified risk aversion parameter *Lambda*. The trading cost minimization is expressed as

$$\min[(MI + PA) + Lambda \cdot TR],$$

where trading costs are market impact *MI*, price appreciation *PA*, and timing risk *TR*. For details, see `marketImpact`, `priceAppreciation`, and `timingRisk`. This example finds a local minimum for this expression. For details about searching for the global minimum, see "Optimization Troubleshooting and Tips".

Here, you can optimize the trade time trade strategy. To optimize percentage of volume and trade schedule strategies, see "Optimize Percentage of Volume Trading Strategy" on page 6-23 and "Optimize Trade Schedule Trading Strategy" on page 6-29.

To access the example code, enter `edit KRGSingleStockOptimizationExample.m` at the command line.

Retrieve Market-Impact Parameters and Create Example Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`.

```
k = krg(miData);
```

Create Single Stock Data

The structure `tradeData` contains data for a single stock. Use a structure or table to define this data. The fields are:

- Number of shares
- Average daily volume
- Volatility
- Stock price
- Initial trade time trade strategy
- Alpha estimate

```
tradeData.Shares = 100000;
tradeData.ADV = 1000000;
tradeData.Volatility = 0.25;
```

```
tradeData.Price = 35;
tradeData.TradeTime = 0.5;
tradeData.Alpha_bp = 50;
```

Define Optimization Parameters

Define risk aversion level `Lambda`. Set `Lambda` from 0 to `Inf`.

```
Lambda = 1;
```

Define lower `LB` and upper `UB` bounds of strategy input for optimization.

```
LB = 0;
UB = 1;
```

Define the function handle `fun` for the objective function. To access the code for this function, enter `edit krgSingleStockOptimizer.m`.

```
fun = @(tradetime)krgSingleStockOptimizer(tradetime,k,tradeData,Lambda);
```

Minimize Trading Costs for Trade Strategy

Minimize the trading costs for the trade time trade strategy. `fminbnd` finds the optimal value for the trade time trade strategy based on the lower and upper bound values. `fminbnd` finds a local minimum for the trading cost minimization expression.

```
[tradeData.TradeTime,totalcost] = fminbnd(fun,LB,UB);
```

Display the optimized trade strategy `tradeData.TradeTime`.

```
tradeData.TradeTime
```

```
ans =

    0.19
```

Estimate Trading Costs for Optimized Strategy

Estimate the trading costs `tradeTimeCosts` using the optimized trade strategy.

```
mi = marketImpact(k,tradeData);
tr = timingRisk(k,tradeData);
pa = priceAppreciation(k,tradeData);
tradeTimeCosts = [totalcost mi pa tr];
```

Display trading costs.

```
tradeTimeCosts
```

```
tradeTimeCosts =

        100.04         56.15          4.63         39.27
```

The trading costs are:

- Total cost
- Market impact
- Price appreciation

- Timing risk

For details about the preceding calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. “Algorithmic Trading Strategies.” Ph.D. Thesis. Fordham University, May 2006.

[2] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[3] Glantz, Morton, and Robert Kissell. *Multi-Asset Risk Modeling*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`fminbnd` | `marketImpact` | `priceAppreciation` | `timingRisk` | `krg`

Related Examples

- “Optimize Percentage of Volume Trading Strategy” on page 6-23
- “Optimize Trade Schedule Trading Strategy” on page 6-29
- “Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17
- “Estimate Portfolio Liquidation Costs” on page 6-20

Optimize Trade Schedule Trading Strategy

This example shows how to optimize the strategy for a single stock by minimizing trading costs using transaction cost analysis from the Kissell Research Group. The optimization minimizes trading costs associated with the trade schedule trading strategy and a specified risk aversion parameter *Lambda*. The trading cost minimization is expressed as

$$\min[(MI + PA) + Lambda \cdot TR],$$

where trading costs are market impact *MI*, price appreciation *PA*, and timing risk *TR*. For details, see `marketImpact`, `priceAppreciation`, and `timingRisk`.

This example requires an Optimization Toolbox™ license. For background information, see "Optimization Theory Overview" (Optimization Toolbox).

Here, you can optimize the trade schedule trade strategy. The optimization finds a local minimum for this expression. For ways to search for the global minimum, see "Local vs. Global Optima" (Optimization Toolbox). To optimize percentage of volume and trade time strategies, see "Optimize Percentage of Volume Trading Strategy" on page 6-23 and "Optimize Trade Time Trading Strategy" on page 6-26.

To access the example code, enter `edit KRGSingleStockOptimizationExample.m` at the command line.

Retrieve Market-Impact Parameters

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`.

```
k = krg(miData);
```

Create Single Stock Data

The structure `tradeData` contains data for a single stock. Use a structure or table to define this data. The fields are:

- Number of shares
- Average daily volume
- Volatility
- Stock price
- Alpha estimate

```
tradeData.Shares = 100000;
tradeData.ADV = 1000000;
```

```
tradeData.Volatility = 0.25;
tradeData.Price = 35;
tradeData.Alpha_bp = 50;
```

Define the number of trades and the volume per trade for the initial strategy. The fields `VolumeProfile` and `TradeSchedule` define the initial trade schedule trade strategy.

```
numIntervals = 26;
tradeData.VolumeProfile = ones(1,numIntervals) * ...
    tradeData.ADV/numIntervals;
tradeData.TradeSchedule = ones(1,numIntervals) .* ...
    (tradeData.Shares./numIntervals);
```

Define Optimization Parameters

Define risk aversion level `Lambda`. Set `Lambda` from 0 to `Inf`.

```
Lambda = 1;
```

Define lower `LB` and upper `UB` bounds of shares traded per interval for optimization.

```
LB = zeros(1,numIntervals);
UB = ones(1,numIntervals) .* tradeData.Shares;
```

Specify constraints `Aeq` and `Beq` to denote that shares traded in the trade schedule must match the total number of shares.

```
Aeq = ones(1,numIntervals);
Beq = tradeData.Shares;
```

Define the maximum number of function evaluations and iterations for optimization. Set `'MaxFunEvals'` and `'MaxIter'` to large values so that the optimization can iterate many times to find a local minimum.

```
options = optimoptions('fmincon','MaxFunEvals',100000,'MaxIter',100000);
```

Define the function handle `fun` for the objective function. To access the code for this function, enter `edit krgSingleStockOptimizer.m`.

```
fun = @(tradeschedule)krgSingleStockOptimizer(tradeschedule,k, ...
    tradeData,Lambda);
```

Minimize Trading Costs for Trade Strategy

Minimize the trading costs for the trade schedule trade strategy. `fmincon` finds the optimal value for the trade schedule trade strategy based on the lower and upper bound values. It does this by finding a local minimum for the trading cost.

```
[tradeData.TradeSchedule,totalcost,exitflag] = fmincon(fun, ...
    tradeData.TradeSchedule,[],[],Aeq,Beq,LB,UB,[],options);
```

To check whether `fmincon` found a local minimum, display the reason why the function stopped.

```
exitflag
```

```
exitflag =

     1.00
```

`fmincon` returns `1` when it finds a local minimum. For details, see `exitflag`.

Display the optimized trade strategy `tradeData.TradeSchedule`.

```
tradeData.TradeSchedule
```

```
ans =

  Columns 1 through 5

     35563.33      18220.14      11688.59       8256.81       6057.39

  ...
```

Estimate Trading Costs for Optimized Strategy

Estimate trading costs `tradeScheduleCosts` using the optimized trade strategy.

```
mi = marketImpact(k,tradeData);
pa = priceAppreciation(k,tradeData);
tr = timingRisk(k,tradeData);
tradeScheduleCosts = [totalcost mi pa tr];
```

Display trading costs.

```
tradeScheduleCosts
```

```
tradeScheduleCosts =

         97.32         47.66          6.75         42.91
```

The trading costs are:

- Total cost
- Market impact
- Price appreciation
- Timing risk

For details about the preceding calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. “Algorithmic Trading Strategies.” Ph.D. Thesis. Fordham University, May 2006.

[2] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[3] Glantz, Morton, and Robert Kissell. *Multi-Asset Risk Modeling*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`fmincon` | `optimoptions` | `marketImpact` | `priceAppreciation` | `timingRisk` | `krg`

Related Examples

- “Optimize Percentage of Volume Trading Strategy” on page 6-23
- “Optimize Trade Time Trading Strategy” on page 6-26
- “Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17
- “Estimate Portfolio Liquidation Costs” on page 6-20

Estimate Trading Costs for Collection of Stocks

This example shows how to estimate four different trading costs for a collection of stocks using Kissell Research Group transaction cost analysis.

Retrieve Market-Impact Parameters and Load Transaction Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`.

```
k = krg(miData);
```

Load the example data `TradeData` from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox.

```
load KRGExampleData.mat TradeData
```

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Estimate Trading Costs

Estimate instantaneous trading cost `itc` using `TradeData`.

```
itc = iStar(k,TradeData);
```

Estimate market-impact cost `mi`.

```
mi = marketImpact(k,TradeData);
```

Estimate timing risk `tr`.

```
tr = timingRisk(k,TradeData);
```

Estimate price appreciation `pa`.

```
pa = priceAppreciation(k,TradeData);
```

See Also

`iStar` | `marketImpact` | `timingRisk` | `priceAppreciation` | `krg`

More About

- “Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17
- “Estimate Portfolio Liquidation Costs” on page 6-20

- “Optimize Percentage of Volume Trading Strategy” on page 6-23
- “Kissell Research Group Data Sets” on page 6-7

Conduct Back Test on Portfolio

This example shows how to conduct a back test on a set of stocks using transaction cost analysis from the Kissell Research Group.

- Analyze the implementation of an investment strategy on a specific day or date range.
- Estimate historical market-impact costs and the corresponding dollar values for the specified historical dates.
- Analyze the trading costs of different orders on various dates.

To access the example code, enter `edit KRGBackTestingExample.m` at the command line.

Retrieve Market-Impact Parameters and Load Historical Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`. Specify initial settings for the date, market-impact code, and number of trading days.

```
k = krg(miData,datetime('today'),1,250);
```

Load the example data `TradeDataBackTest` from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox.

```
load KRGExampleData TradeDataBackTest
```

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Prepare Data for Back Testing

Determine the number of stocks `numRecords` in the portfolio.

```
numRecords = length(TradeDataBackTest.Symbol);
```

Preallocate the output data table `o`.

```
o = table(TradeDataBackTest.Symbol,TradeDataBackTest.Side, ...
    TradeDataBackTest.Date,NaN(numRecords,1),NaN(numRecords,1), ...
    'VariableNames',{'Symbol','Side','Date','MI','MIDollar'});
```

Ensure that the number of shares is a positive value using the `abs` function.

```
TradeDataBackTest.Shares = abs(TradeDataBackTest.Shares);
```

Convert trade time trade strategy to the percentage of volume trade strategy.

```
TradeDataBackTest.TradeTime = TradeDataBackTest.TradeTime ...
    .* TradeDataBackTest.ADV;
```

```
TradeDataBackTest.POV = krg.tradetime2pov(TradeDataBackTest.TradeTime, ...
    TradeDataBackTest.Shares);
```

Conduct Back Test by Estimating Historical Market-Impact Costs

Estimate the historical market-impact costs for each stock in the portfolio on different dates using `marketImpact`. Convert market-impact cost from decimal into local dollars. Retrieve the resulting data in the output data table `o`.

```
for ii = 1:numRecords

    k.MiDate = TradeDataBackTest.Date(ii);
    k.MiCode = TradeDataBackTest.MICode(ii);

    o.MI(ii) = marketImpact(k,TradeDataBackTest(ii,:));
    MIDollars = (TradeDataBackTest.Shares(ii) * TradeDataBackTest.Price(ii)) ...
        * o.MI(ii)/10000 * TradeDataBackTest.FXRate(ii);

    o.MIDollar(ii) = MIDollars;

end
```

Display the first three rows of output data.

```
o(1:3,:)
```

```
ans =

    Symbol    Side       Date         MI     MIDollar
    ______    ____    __________    ____    ________

    'A'       1.00    '5/1/2015'    1.04     103.91
    'B'       1.00    '5/1/2015'    3.09    3864.44
    'C'       1.00    '5/1/2015'    8.54    5335.03
```

The output data contains these variables:

- Stock symbol
- Side
- Historical trade date
- Historical market-impact cost in basis points
- Historical market-impact value in local dollars

References

[1] Kissell, Robert. “Creating Dynamic Pre-Trade Models: Beyond the Black Box.” *Journal of Trading*. Vol. 6, Number 4, Fall 2011, pp. 8–15.

[2] Kissell, Robert. “TCA in the Investment Process: An Overview.” *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[3] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Chung, Grace and Robert Kissell. “An Application of Transaction Costs in the Portfolio Optimization Process.” *Journal of Trading*. Vol. 11, Number 2, Spring 2016, pp. 11–20.

See Also

`krg` | `marketImpact`

More About

- “Conduct Stress Test on Portfolio” on page 6-38
- “Liquidate Dollar Value from Portfolio” on page 6-43
- “Kissell Research Group Data Sets” on page 6-7

Conduct Stress Test on Portfolio

This example shows how to conduct a stress test on a set of stocks using transaction cost analysis from the Kissell Research Group.

- Estimate historical market-impact costs and the corresponding dollar values for the specified date range.
- Use trading costs to screen stocks in a portfolio and estimate the cost to liquidate or purchase a specified number of shares.
- Analyze trading costs during volatile periods of time such as a financial crisis, flash crash, or debt crisis.

To access the example code, enter `edit KRGStressTestingExample.m` at the command line.

Retrieve Market-Impact Parameters and Load Historical Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Load the example data `TradeDataStressTest` from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox.

```
load KRGExampleData TradeDataStressTest
```

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Create a Kissell Research Group transaction cost analysis object `k`. Specify initial settings for the date, market-impact code, and number of trading days.

```
k = krg(miData,datetime('today'),1,250);
```

Prepare Data for Stress Testing

Specify the date range from May 1, 2015, through July 31, 2015.

```
startDate = '5/1/2015';
endDate = '7/31/2015';
```

Determine the number of stocks `numStocks` in the portfolio. Create a date range `dateRange` from the specified dates. Find the number of days `numDates` in the date range.

```
numStocks = length(TradeDataStressTest.Symbol);
dateRange = (datenum(startDate):datenum(endDate))';
numDates = length(dateRange);
```

Preallocate the output data table `o`.

```
outLength = numStocks*numDates;
symbols = TradeDataStressTest.Symbol(:,ones(1,numDates));
sides = TradeDataStressTest.Side(:,ones(1,numDates));
dates = dateRange(:,ones(1,numStocks))';

o = table(symbols(:),sides(:),dates(:),NaN(outLength,1),NaN(outLength,1), ...
    'VariableNames',{'Symbol','Side','Date','MI','MIDollar'});
```

Ensure that the number of shares is a positive value using the `abs` function.

```
TradeDataStressTest.Shares = abs(TradeDataStressTest.Shares);
```

Convert trade time trade strategy to the percentage of volume trade strategy.

```
TradeDataStressTest.TradeTime = TradeDataStressTest.TradeTime ...
    .* TradeDataStressTest.ADV;
TradeDataStressTest.POV = krg.tradetime2pov(TradeDataStressTest.TradeTime, ...
    TradeDataStressTest.Shares);
```

Conduct Stress Test by Estimating Historical Market-Impact Costs

Estimate the historical market-impact costs for each stock in the portfolio for the date range using `marketImpact`. Convert market-impact cost from decimal into local dollars. Retrieve the resulting data in the output data table `o`.

```
kk = 1;
for ii = dateRange(1):dateRange(end)

    for jj = 1:numStocks

        k.MiCode = TradeDataStressTest.MICode(jj);
        k.MiDate = ii;

        o.MI(kk) = marketImpact(k,TradeDataStressTest(jj,:));
        o.MIDollar(kk) = (TradeDataStressTest.Shares(jj) ...
            * TradeDataStressTest.Price(jj)) ...
            * o.MI(kk) /10000 * TradeDataStressTest.FXRate(jj);

        kk = kk + 1;

    end

end
```

Display the first three rows of output data.

```
o(1:3,:)
```

```
ans =

    Symbol    Side      Date         MI      MIDollar
    ______    ____    _________    _____    ________

    'A'       1.00    736085.00     3.84      384.31
    'B'       1.00    736085.00    11.43    14292.24
    'C'       1.00    736085.00    32.69    20430.65
```

The output data contains these variables:

- Stock symbol
- Side

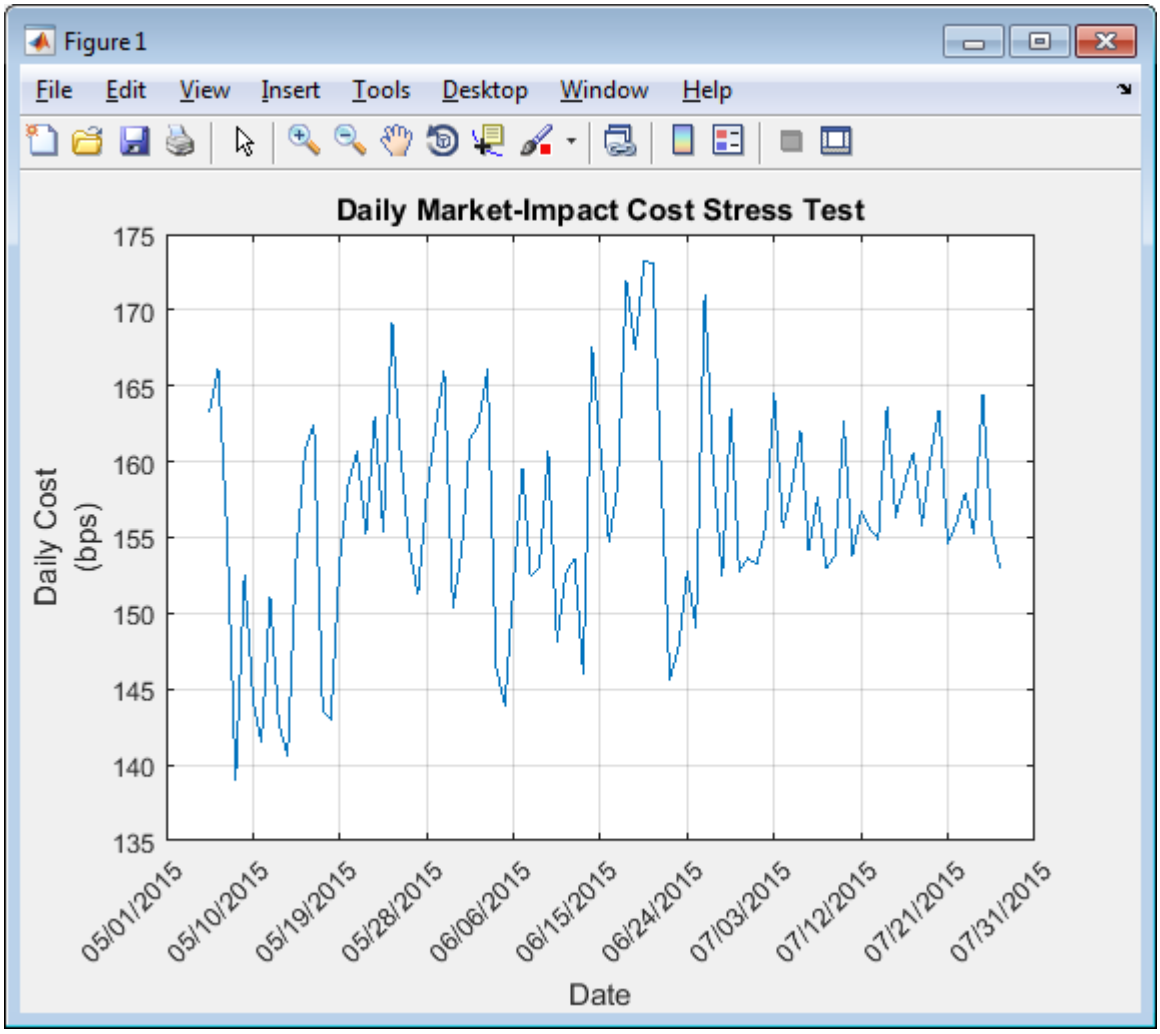
- Historical trade date
- Historical market-impact cost in basis points
- Historical market-impact value in local dollars

Retrieve the daily market-impact cost `dailyCost`. Determine the number of days `numDays` in the output data. Loop through the data and sum the market-impact costs for individual stocks for each day.

```
numDays = length(o.Date)/numStocks;

idx = 1;
for i = 1:numDays

  dailyCost.Date(i) = o.Date(idx);
  dailyCost.DailyMiCost(i) = sum(o.MI(idx:idx+(numStocks-1)));
  idx = idx+numStocks;

end
```

Display the daily market-impact cost in the specified date range. This figure demonstrates how market-impact costs change over time.

```
plot(b.Date,b.DailyMiCost)
ylabel({'Daily Cost','(bps)'})
title('Daily Market-Impact Cost Stress Test')
xlabel('Date')
grid on
xData = linspace(b.Date(1),b.Date(92),11);
a = gca;
a.XAxis.TickLabels = datestr(xData,'mm/dd/yyyy');
a.XTickLabelRotation = 45;
```



References

[1] Kissell, Robert. “Creating Dynamic Pre-Trade Models: Beyond the Black Box.” *Journal of Trading*. Vol. 6, Number 4, Fall 2011, pp. 8–15.

[2] Kissell, Robert. “TCA in the Investment Process: An Overview.” *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[3] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Chung, Grace and Robert Kissell. “An Application of Transaction Costs in the Portfolio Optimization Process.” *Journal of Trading*. Vol. 11, Number 2, Spring 2016, pp. 11–20.

See Also

`krg` | `marketImpact`

More About

- “Conduct Back Test on Portfolio” on page 6-35
- “Liquidate Dollar Value from Portfolio” on page 6-43
- “Kissell Research Group Data Sets” on page 6-7

Liquidate Dollar Value from Portfolio

This example shows how to liquidate a dollar value from a portfolio while minimizing market-impact costs using transaction cost analysis from the Kissell Research Group. This example always results in a portfolio that shrinks in size. The market-impact cost minimization is expressed as

$$\underset{x}{\operatorname{argmin}}[MI'|x|],$$

where MI is the market-impact cost for the traded shares and x denotes the final weights for each stock.

This example requires an Optimization Toolbox license. For background information, see "Optimization Theory Overview" (Optimization Toolbox).

The optimization finds a local minimum for the market-impact cost of liquidating a dollar value from a portfolio. For ways to search for the global minimum, see "Local vs. Global Optima" (Optimization Toolbox).

To access the example code, enter `edit KRGLiquidityOptimizationExample.m` at the command line.

Retrieve Market-Impact Parameters and Load Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`. Specify initial settings for the date, market-impact code, and number of trading days.

```
k = krg(miData,datetime('today'),1,250);
```

Load the example data `TradeDataPortOpt` and the covariance data `CovarianceData` from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox. Limit the data set to the first 10 rows.

```
load KRGExampleData.mat TradeDataPortOpt CovarianceData

n = 10;
TradeDataPortOpt = TradeDataPortOpt(1:n,:);
CovarianceData = CovarianceData(1:n,1:n);
C = table2array(CovarianceData);
```

For a description of the example data, see "Kissell Research Group Data Sets" on page 6-7.

Define Optimization Parameters

Set the portfolio liquidation value to $100,000,000. Set the portfolio risk boundaries between 90% and 110%. Set the maximum total market-impact cost to 50 basis points. Determine the number of

stocks in the portfolio. Retrieve the upper bound constraint for the maximum market-impact cost for liquidating shares in each stock.

```
PortLiquidationValue = 100000000;
PortRiskBounds = [0.9 1.10];
maxTotalMI = 0.005;
numPortStocks = length(TradeDataPortOpt.Symbol);
maxMI = TradeDataPortOpt.UB_MaxMI;
```

Determine the target portfolio value `PortfolioTargetValue` by subtracting the portfolio liquidation value from the total portfolio value.

```
PortfolioValue = sum(TradeDataPortOpt.Value);
absPortValue = abs(TradeDataPortOpt.Value);
PortfolioAbsValue = sum(absPortValue);
PortfolioTargetValue = PortfolioValue-PortLiquidationValue;
```

Determine the current portfolio weight `w` based on the value of each stock in the portfolio.

```
w = sign(TradeDataPortOpt.Shares).*absPortValue/PortfolioAbsValue;
```

Specify constraints `Aeq` and `beq` to indicate that the weights must sum to one. Initialize the linear inequality constraints `A` and `b`.

```
Aeq = ones(1,numPortStocks);
beq = 1;

A = [];
b = [];
```

Retrieve the lower and upper bounds for the final portfolio weight in `TradeDataPortOpt`.

```
LB = TradeDataPortOpt.LB_Wt;
UB = TradeDataPortOpt.UB_Wt;
```

Determine the lower and upper bounds for the number of shares in the final portfolio using other optional constraints in the example data set.

```
lbShares = max([TradeDataPortOpt.LB_MinShares, ...
    TradeDataPortOpt.LB_MinValue./TradeDataPortOpt.Price, ...
    TradeDataPortOpt.LB_MinPctADV.*TradeDataPortOpt.ADV],[],2);

ubShares = min([TradeDataPortOpt.UB_MaxShares, ...
    TradeDataPortOpt.UB_MaxValue./TradeDataPortOpt.Price, ...
    TradeDataPortOpt.UB_MaxPctADV.*TradeDataPortOpt.ADV],[],2);
```

Specify the initial portfolio weights.

```
x0 = TradeDataPortOpt.Value./sum(TradeDataPortOpt.Value);
x = x0;
```

Define optimization options. Set the optimization algorithm to sequential quadratic programming. Set the termination tolerance on the function value and on `x`. Set the tolerance on the constraint violation. Set the termination tolerance on the PCG iteration. Set the maximum number of function evaluations `'MaxFunEvals'` and iterations `'MaxIter'`. The options `'MaxFunEvals'` and `'MaxIter'` are set to large values so that the optimization can iterate many times to find a local minimum. Set the minimum change in variables for finite differencing.

```
options = optimoptions('fmincon','Algorithm','sqp', ...
    'TolFun',10E-8,'TolX',10E-16,'TolCon',10E-8,'TolPCG',10E-8, ...
    'MaxFunEvals',50000,'MaxIter',50000,'DiffMinChange',10E-8);
```

Minimize Market-Impact Costs for Portfolio Liquidation

Define the function handle `objectivefun` for the sample objective function `krgLiquidityFunction`. To access the code for this function, enter `edit krgLiquidityFunction.m`. Define the function handle `constraintsfun` for the sample function `krgLiquidityConstraint` that sets additional constraints. To access the code for this function, enter `edit krgLiquidityConstraint.m`.

```
objectivefun = @(x) krgLiquidityFunction(x,TradeDataPortOpt, ...
    PortfolioTargetValue,k);

constraintsfun = @(x) krgLiquidityConstraint(x,w,C,TradeDataPortOpt, ...
    PortfolioTargetValue,PortRiskBounds,lbShares,ubShares,maxMI,maxTotalMI,k);
```

Minimize the market-impact costs for the portfolio liquidation. `fmincon` finds the optimal value for the portfolio weight for each stock based on the lower and upper bound values. It does this by finding a local minimum for the market-impact cost.

```
[x,~,exitflag] = fmincon(objectivefun,x0,A,b,Aeq,beq,LB,UB, ...
    constraintsfun,options);
```

To check whether `fmincon` found a local minimum, display the reason why the function stopped.

```
exitflag
```

```
exitflag =

     1.00
```

`fmincon` returns `1` when it finds a local minimum. For details, see `exitflag`.

Determine the optimized weight value `x1` of each stock in the portfolio in decimal format.

```
x1 = x.*PortfolioTargetValue/PortfolioValue;
```

Determine the optimized portfolio target value `TargetValue` and number of shares `SharesToTrade` for each stock in the portfolio.

```
TargetShares = x*PortfolioTargetValue./TradeDataPortOpt.Price;
SharesToTrade = TradeDataPortOpt.Shares-TargetShares;
TargetValue = x*PortfolioTargetValue;
TradeDataPortOpt.Shares = abs(SharesToTrade);
```

Determine the optimized percentage of volume strategy.

```
TradeDataPortOpt.TradeTime = TradeDataPortOpt.TradeTime ...
    .* TradeDataPortOpt.ADV;
TradeDataPortOpt.POV = krg.tradetime2pov(TradeDataPortOpt.TradeTime, ...
    TradeDataPortOpt.Shares);
```

Estimate the market-impact costs `MI` for the number of shares to liquidate.

```
MI = marketImpact(k,TradeDataPortOpt)/10000;
```

To view the market-impact cost in decimal format, specify the display format. Display the market-impact cost for the first three stocks in the portfolio.

```
format

MI(1:3)

ans =

   1.0e-03 *

    0.1477
    0.1405
    0.1405
```

To view the target number of shares with two decimal places, specify the display format. Display the target number of shares for the first three stocks in the portfolio.

```
format bank

TargetShares(1:3)

ans =

     -23640.11
    -154656.73
     -61193.04
```

The negative values denote selling shares from the portfolio.

Display the traded value for the first three stocks in the portfolio.

```
TargetValue(1:3)

ans =

    -968062.45
   -1521760.41
   -2448131.64
```

To simulate trading the target number of shares on a historical date range, you can now conduct a stress test on the optimized portfolio. For details about conducting a stress test, see “Conduct Stress Test on Portfolio” on page 6-38.

References

[1] Kissell, Robert. “Creating Dynamic Pre-Trade Models: Beyond the Black Box.” *Journal of Trading*. Vol. 6, Number 4, Fall 2011, pp. 8–15.

[2] Kissell, Robert. “TCA in the Investment Process: An Overview.” *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[3] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Chung, Grace and Robert Kissell. “An Application of Transaction Costs in the Portfolio Optimization Process.” *Journal of Trading*. Vol. 11, Number 2, Spring 2016, pp. 11–20.

See Also

`krg` | `marketImpact` | `fmincon` | `optimoptions`

More About

- “Conduct Stress Test on Portfolio” on page 6-38
- “Optimize Trade Schedule Trading Strategy” on page 6-29
- “Optimize Long Portfolio” on page 6-48
- “Kissell Research Group Data Sets” on page 6-7

Optimize Long Portfolio

This example shows how to determine the optimal portfolio weights for a specified dollar value using transaction cost analysis from the Kissell Research Group. The sample portfolio contains only long shares of stock. You can incorporate risk, return, and market-impact cost during implementation of the investment decision.

This example requires an Optimization Toolbox license. For background information, see "Optimization Theory Overview" (Optimization Toolbox).

The `KRGPortfolioOptimizationExample` function, which you can access by entering `edit KRGPortfolioOptimizationExample.m`, addresses three different optimization scenarios:

1. Maximize the trade off between net portfolio return and portfolio risk. The trade off maximization is expressed as

   $$\underset{x}{\text{argmax}}[R'x - MI'|x| - \lambda x'Cx],$$

   where:

   - R is the estimated return for each stock in the portfolio.
   - x denotes the weights for each stock in the portfolio.
   - MI is the market-impact cost for the specified dollar value and share quantities.
   - λ is the specified risk aversion parameter.
   - C is the covariance matrix of the stock data.
2. Minimize the portfolio risk subject to a minimum return target using

   $$\underset{x}{\text{argmin}}[x'Cx].$$
3. Maximize net portfolio return subject to a maximum risk exposure target using

   $$\underset{x}{\text{argmax}}[R'x - MI'|x|].$$

Lower and upper bounds constrain x in each scenario. Each optimization finds a local optimum. For ways to search for the global optimum, see "Local vs. Global Optima" (Optimization Toolbox).

Retrieve Market-Impact Parameters and Load Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`. Specify initial settings for the date, market-impact code, and number of trading days.

```
k = krg(miData,datetime('today'),1,250);
```

Load the example data `TradeDataPortOpt` and the covariance data `CovarianceData` from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox. Limit the data set to the first 50 rows.

```
load KRGExampleData TradeDataPortOpt CovarianceData

n = 50;
TradeDataPortOpt = TradeDataPortOpt(1:n,:);
CovarianceData = CovarianceData(1:n,1:n);
```

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Maximize Net Portfolio Return

Run the optimization scenario using the example and covariance data. To run the first optimization, specify `1` in the last input argument.

```
[Weight,Shares,Value,MI] = KRGPortfolioOptimizationExample(TradeDataPortOpt, ...
    CovarianceData,1);
```

`KRGPortfolioOptimizationExample` returns the optimized values for each stock in the portfolio:

- Portfolio weight
- Number of shares
- Portfolio dollar value
- Market-impact cost

To run the other two scenarios, specify `2` or `3` in the last input argument of `KRGPortfolioOptimizationExample`.

Display the portfolio weight for the first three stocks in the portfolio in decimal format.

```
format

Weight(1:3)

ans =

    0.0100
    0.3198
    0.1610
```

Display the number of shares using two decimal places for the first three stocks in the portfolio.

```
format bank

Shares(1:3)

ans =

      24420.02
    3249893.71
     402364.47
```

Display the portfolio dollar value for the first three stocks in the portfolio.

```
Value(1:3)

ans =

    1000000.00
   31977654.17
   16097274.50
```

Display the market-impact cost for the first three stocks in the portfolio in decimal format.

```
format

MI(1:3)

ans =

   1.0e-03 *

    0.1250
    0.7879
    0.3729
```

References

[1] Kissell, Robert. “Creating Dynamic Pre-Trade Models: Beyond the Black Box.” *Journal of Trading*. Vol. 6, Number 4, Fall 2011, pp. 8–15.

[2] Kissell, Robert. “TCA in the Investment Process: An Overview.” *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[3] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[4] Chung, Grace and Robert Kissell. “An Application of Transaction Costs in the Portfolio Optimization Process.” *Journal of Trading*. Vol. 11, Number 2, Spring 2016, pp. 11–20.

See Also

`krg` | `marketImpact` | `fmincon` | `optimoptions`

More About

- “Optimize Trade Schedule Trading Strategy” on page 6-29
- “Liquidate Dollar Value from Portfolio” on page 6-43
- “Kissell Research Group Data Sets” on page 6-7

Determine Buy-Sell Imbalance Using Cost Index

This example shows how to determine buy-sell imbalance using transaction cost analysis from the Kissell Research Group. Imbalance is the difference between buy-initiated and sell-initiated volume given actual market conditions on the day and over the specified trading period. A positive imbalance indicates buying pressure in the stock and a negative imbalance indicates selling pressure. The cost index helps investors to understand how the trading cost environment affects the order flow in the market. The index can be a performance-based index, such as the S&P 500, that shows market movement and value or a volatility index that shows market uncertainty.

The imbalance share quantity is the value of x such that

$$0 = |MICost| \cdot 10000 - MI(x),$$

where

$$MI(x) = \left[b_1 \cdot \left(\frac{x}{Volume}\right)^{a4} + (1 - b_1)\right] \cdot \left[a_1 \cdot \left(\frac{x}{ADV}\right)^{a2} \cdot \sigma^{a3} \cdot Price^{a5}\right].$$

MI is the market-impact cost for a stock transaction. The estimated trading costs represent the incremental price movement of the stock in relation to the underlying index price movement. *Volume* is the actual daily volume of a stock in the basket. *ADV* is the average daily volume of a stock in the basket. *Price* is the price of a stock in the basket. The other variables in the equations are:

- σ — Price volatility.
- a_1 — Price sensitivity to order flow.
- a_2 — Order size shape.
- a_3 — Volatility shape.
- a_4 — Percentage of volume rate shape.
- a_5 — Price shape.
- $1 - b_1$ — Percentage of permanent market impact. Permanent impact is the unavoidable impact cost that occurs because of the information content of the trade.
- b_1 — Percentage of temporary market impact. Temporary impact is dependent upon the trading strategy. Temporary impact occurs because of the liquidity demands of the investor.
- $MICost = TotalCost - Beta \cdot IndexCost$, where:
  - *TotalCost* — Change in the volume-weighted average price compared to the open price for the stocks.
  - *Beta* — Beta.
  - *IndexCost* — Change in the volume-weighted average price compared to the open price for the index. Index cost adjusts the price for market movement using an underlying index and beta.

In this example, you can run this code using current or historical data. Current data includes prices starting from the open time through the current time. Historical data uses the prices over the entire day. Historical costs use market-impact parameters for the specified region and date. Therefore, historical costs change from record to record.

For a current cost index, you load the example table `TradeDataCurrent` from the file `KRGExampleData.mat`. For a historical cost index, you load the example table

`TradeDataHistorical` from the file `KRGExampleData.mat`. This example calculates a current cost index.

To access the example code, enter `edit KRGCostIndexExample.m` at the command line.

After running this code, you can submit an order for execution using Bloomberg, for example.

This example requires an Optimization Toolbox license. For background information, see "Optimization Theory Overview" (Optimization Toolbox).

Retrieve Market-Impact Parameters and Load Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`. Specify initial settings for the date, market-impact code, and number of trading days.

```
k = krg(miData,datetime('today'),1,250);
```

Load the example data `TradeDataCurrent`, which is included with the Datafeed Toolbox. Calculate the number of stocks in the portfolio.

```
load KRGExampleData.mat TradeDataCurrent
TradeData = TradeDataCurrent;

numStocks = height(TradeData);
```

For a description of the example data, see "Kissell Research Group Data Sets" on page 6-7.

Define Optimization Parameters

Define the maximum number of function iterations for optimization. Set `'MaxIterations'` to a large value so that the optimization can iterate many times to solve a system of nonlinear equations.

```
options = optimoptions('fsolve','MaxIterations',4000);
```

Estimate Trading Costs Using Cost Index

Determine the total cost and beta cost. Calculate the side of the initial market-impact cost estimate. Determine the initial volume `x0`.

```
totalCost = TradeData.VWAP ./ TradeData.Open - 1;
indexCost = TradeData.Beta .* ...
    (TradeData.IndexVWAP ./ TradeData.IndexOpen - 1);
miCost = totalCost - indexCost;
sideIndicator = sign(miCost);
x0 = 0.5 * TradeData.Volume;
```

Create a table that stores all output data. First, add these variables:

- `Symbol` — Stock symbol
- `Date` — Transaction date
- `Side` — Side
- `TotalVolume` — Transaction volume
- `TotalCost` — Total transaction cost
- `IndexCost` — Index cost

```
costIndexTable = table;
costIndexTable.Symbol = TradeData.Symbol;
costIndexTable.Date = TradeData.Date;
costIndexTable.Side = sideIndicator;
costIndexTable.TotalVolume = TradeData.Volume;
costIndexTable.TotalCost = totalCost;
costIndexTable.IndexCost = indexCost;
```

Use a `for`-loop to calculate the cost index for each stock in the portfolio. Each stock might have different market-impact codes and dates. Use the `costIndexExampleEq` function that contains the nonlinear equation to solve. To access the code for the `costIndexExampleEq` function, enter `edit KRGCostIndexExample.m`.

Add these variables to the output table:

- `Imbalance` — Imbalance
- `ImbalancePctADV` — Imbalance as percentage of average daily volume
- `ImbalancePctDayVolume` — Imbalance as percentage of the daily volume
- `BuyVolume` — Buy volume
- `SellVolume` — Sell volume
- `MI` — Market-impact cost
- `ExcessCost` — Excess cost

```
for i = 1:numStocks

  % Set the MiCode and MiDate of the object for each stock
  k.MiCode = TradeData.MICode(i);
  k.MiDate = TradeData.Date(i);

  % Solve for Shares for each stock that results in the target market
  % impact cost.

  % In this example, x is the number of shares (imbalance) that causes
  % the MI impact cost, the number of shares that result in a market
  % impact cost of MI. Here use abs(MI) since market-impact
  % cost is always positive. If the market-impact cost is 0.0050 then
  % fsolve tries to find the number of shares x so that the market
  % impact formula returns 0.0050.
  % Note that fsolve is using the cost in basis points.
  x = fsolve(@(x) costIndexExampleEq(x,miCost(i),TradeData(i,:),k), ...
      x0(i),options);

  % The imbalance must be between 0 and the actual traded volume.
  x = max(min(x,TradeData.Volume(i)),0);

  % Recalculate the percentage of volume and shares based on x.
  TradeData.POV(i) = x/TradeData.Volume(i);
  TradeData.Shares(i) = x;

  % Calculate the new cost as a decimal value.
  mi = marketImpact(k,TradeData(i,:))/10000;
```

```
    % imbalance is the share amount specified as buy or sell by the
    % sideIndicator.
    imbalance = sideIndicator(i) * x;

    % Calculate the buy and sell volumes.
    % Knowing that:
    %
    % Volume = buyVolume + sellVolume;
    % Imbalance = buyVolume - sellVolume;
    %
    % Solve for buyVolume and sellVolume
    buyVolume = (TradeData.Volume(i) + imbalance) / 2;
    sellVolume = (TradeData.Volume(i) - imbalance) / 2;

    % Fill output table
    costIndexTable.Imbalance(i,1) = imbalance;
    costIndexTable.ImbalancePctADV(i,1) = imbalance/TradeData.ADV(i);
    costIndexTable.ImbalancePctDayVolume(i,1) = imbalance/TradeData.Volume(i);
    costIndexTable.BuyVolume(i,1) = buyVolume;
    costIndexTable.SellVolume(i,1) = sellVolume;
    costIndexTable.MI(i,1) = mi * sideIndicator(i);
    costIndexTable.ExcessCost(i,1) = totalCost(i) - mi - indexCost(i);

end
```

Display the imbalance amount for the first stock in the output data.

```
costIndexTable.Imbalance(1)
```

```
ans =

  -8.7894e+04
```

The negative imbalance amount indicates selling pressure. Decide whether to buy, hold, or sell shares of this stock in the portfolio.

For details about the preceding calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[2] Malamut, Roberto. “Multi-Period Optimization Techniques for Trade Scheduling.” Presentation at the QWAFAFEW New York Conference, April 2002.

[3] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`krg` | `marketImpact` | `optimoptions` | `fsolve`

More About

- “Conduct Back Test on Portfolio” on page 6-35
- “Conduct Stress Test on Portfolio” on page 6-38
- “Kissell Research Group Data Sets” on page 6-7

Rank Broker Performance

This example shows how to determine the best performing brokers across transactions using transaction cost analysis from the Kissell Research Group. You rank brokers based on broker value add and arrival cost, and then determine which brokers perform best in which market conditions and trading characteristics. A positive value add indicates that the broker exceeds performance expectations given actual market conditions and trade characteristics, which results in fund savings. A negative value add indicates that the broker did not meet performance expectations, which result in an incremental cost to the fund.

In this example, you find which brokers over- or under-perform by comparing arrival costs and estimated trading costs. A broker with an arrival cost that is less than the estimated trading cost over-performs, which causes the fund to save money. A broker with an arrival cost that is greater than the estimated trading cost under-performs, which causes the fund to incur an incremental cost.

This example also shows how to estimate costs by broker, which requires custom market-impact parameters for each broker.

You can use similar steps as in this example to rank trading venues and algorithms.

To access the example code, enter `edit KRGTradePerformanceRankingExample.m` at the command line.

After executing the code in this example, you can submit an order for execution using Bloomberg, for example.

Retrieve Market-Impact Parameters and Load Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data with broker codes in the `MI_Broker.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Broker.csv');
close(f)

miData = readtable('MI_Broker.csv','delimiter',',', ...
    'ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`. Specify initial settings for the date, market-impact code, and number of trading days.

```
k = krg(miData,datetime('today'),1,250);
```

Load the example data `TradeData`, `Basket`, and `BrokerNames`, which is included with the Datafeed Toolbox.

```
load KRGExampleData.mat TradeData Basket BrokerNames
```

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Calculate Costs for Each Broker

Select the trade categories. Calculate the average arrival cost, market-impact cost, and broker value add for each broker.

```
TradeData.TradeSize = TradeData.Shares ./ TradeData.ADV;
TradeData.ArrivalCost = TradeData.SideIndicator .* ...
    (TradeData.AvgExecPrice ./ TradeData.ArrivalPrice-1) * 10000;
TradeData.MI = marketImpact(k,TradeData);
TradeData.ValueAdd = TradeData.MI - TradeData.ArrivalCost;
```

Retrieve broker names and the number of brokers. Preallocate output data variables.

```
uniqueBrokers = unique(TradeData.Broker);
numBrokers = length(uniqueBrokers);
avgCost = NaN(numBrokers,1);
avgMI = NaN(numBrokers,1);
avgValueAdd = NaN(numBrokers,1);
```

Rank Brokers by Average Broker Value Add

Calculate broker ranking using a transaction size between 5% and 10% of average daily volume (ADV). Calculate average arrival cost, average market-impact cost, and average broker value add.

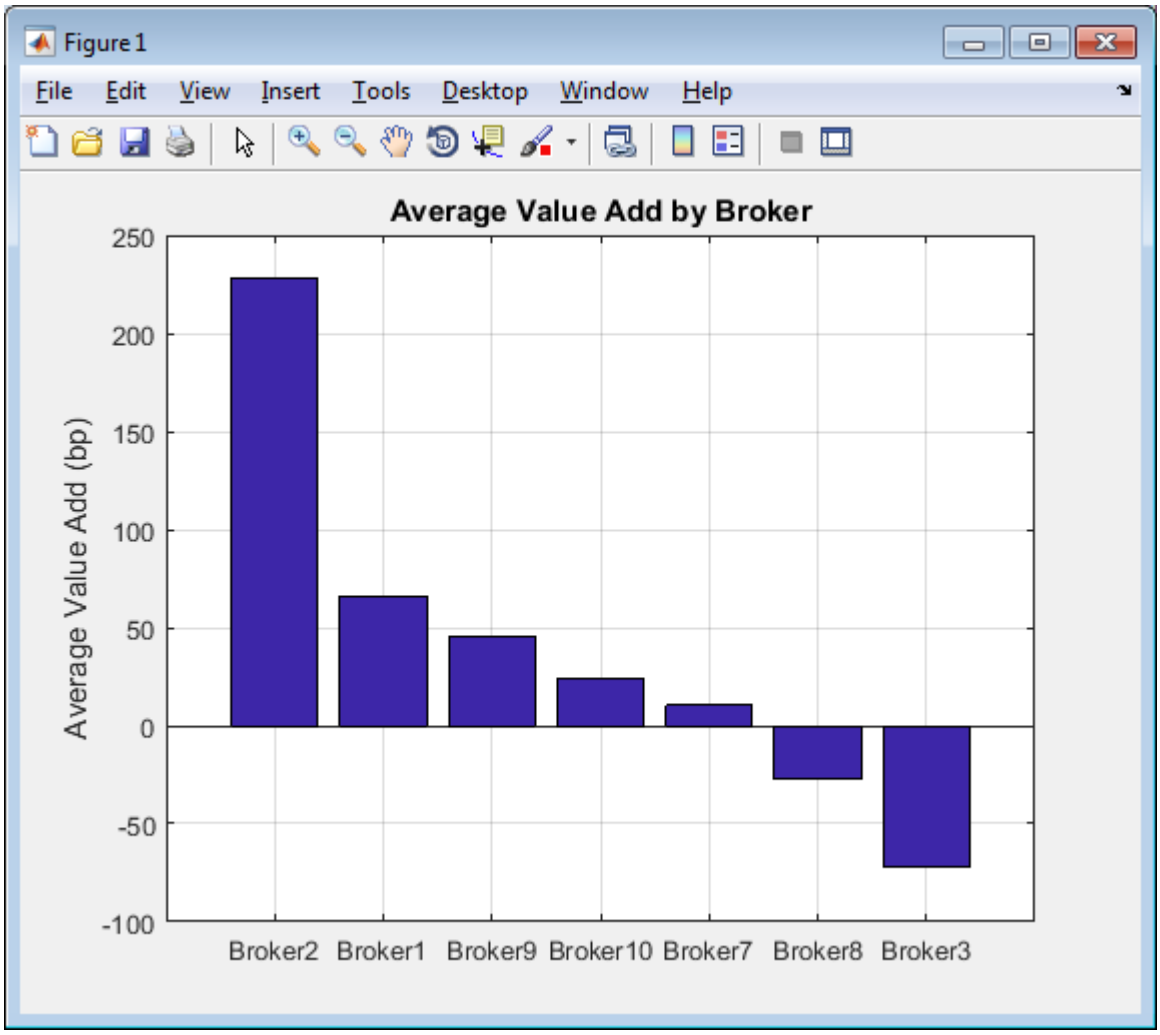
```
indBroker = (TradeData.TradeSize >= 0.05) & (TradeData.TradeSize <= 0.10);

if any(indBroker)
    TD = TradeData(indBroker,:);
    for i = 1:numBrokers
        j = strcmp(TD.Broker,uniqueBrokers(i));
        if any(j)
            avgCost(i) = mean(TD.ArrivalCost(j));
            avgMI(i) = mean(TD.MI(j));
            avgValueAdd(i) = mean(TD.ValueAdd(j));
        end
    end
end

% Get valid average cost values (non NaN's)
indAvgCost = ~isnan(avgCost);
```

Create a table to store the broker ranking. Sort the ranking by average cost.

```
BrokerRankings = table(uniqueBrokers(indAvgCost),(1:sum(indAvgCost))', ...
    avgCost(indAvgCost),avgMI(indAvgCost),avgValueAdd(indAvgCost), ...
    'VariableNames',{'Broker','Rank','AvgArrivalCost','AvgMI','AvgValueAdd'});
BrokerRankings = sortrows(BrokerRankings,-5);
BrokerRankings.Rank = (1:sum(indAvgCost))';  % Reset rank
```

Compare the average broker value add in basis points using a bar graph.

```
bar(BrokerRankings.AvgValueAdd)
set(gca,'XTickLabel',BrokerRankings.Broker)
ylabel('Average Value Add (bp)')
title('Average Value Add by Broker')
grid
```



The broker `Broker2` over-performs while `Broker3` under-performs across transactions. Decide to use `Broker2` for future transactions.

Estimate Trading Costs for Trade List

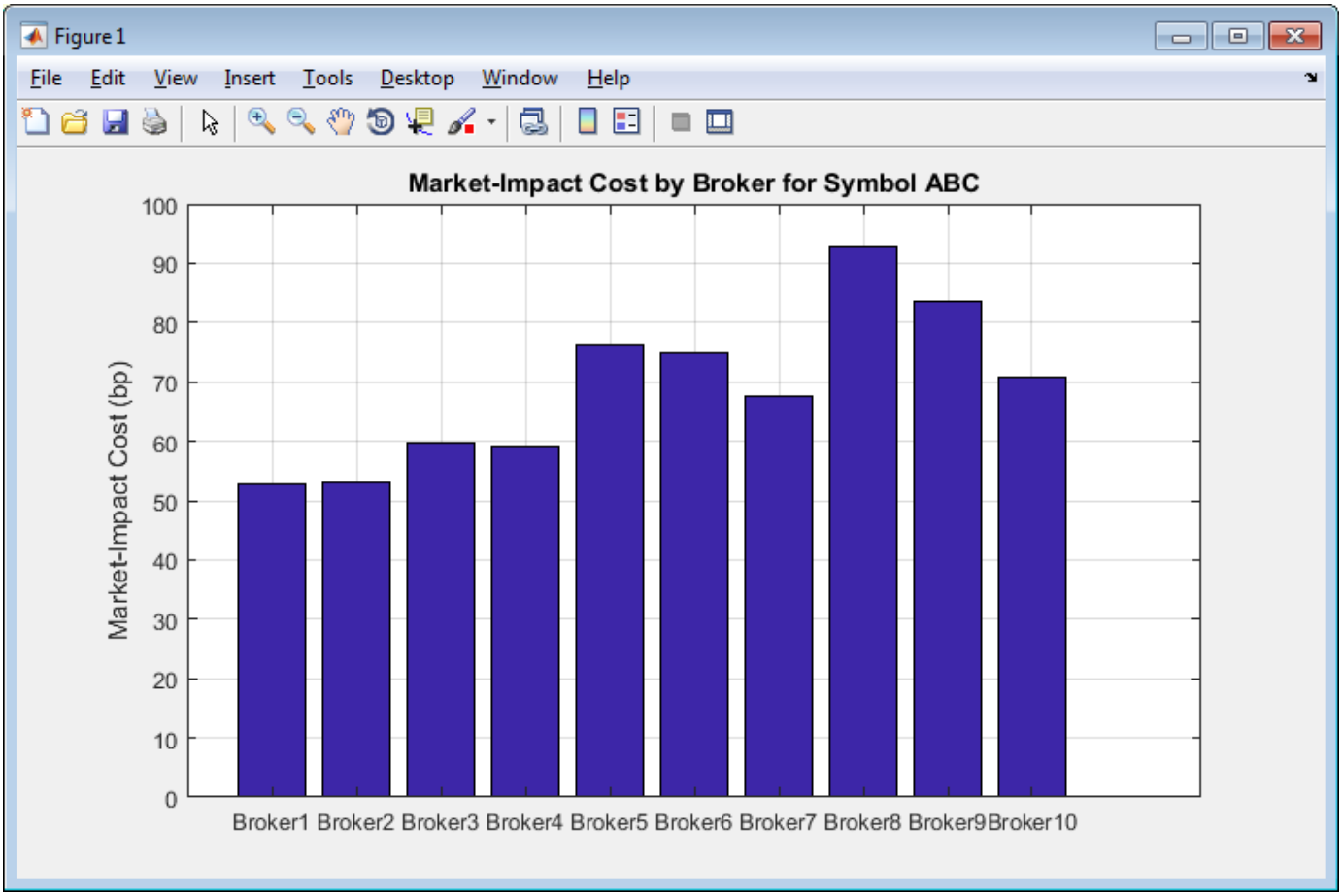
Estimate the trading costs for each broker using a specified order or trade list.

```
% Get the number of orders from the trade list table
numOrders = size(Basket.Symbols,1);

% Calculate pre-trade cost for each broker for each order
BrokerPreTrade = zeros(numOrders,numBrokers);
for i = 1:numBrokers
    % Market-impact code for broker corresponds to the MICode in the market
    % impact data, for example, Broker1 = 1.
    k.MiCode = i;

    % Calculate market-impact cost for each broker
    BrokerPreTrade(:,i) = marketImpact(k,Basket);
end

% Convert output to a table with the symbols used as the row names.
BrokerPreTrade = array2table(BrokerPreTrade,'VariableNames', ...
    BrokerNames.Broker,'RowNames',Basket.Symbols);
```

Compare Market-Impact Costs by Broker

For one stock ABC, compare market-impact cost in basis points for each broker using a bar graph.

```
% Plot best broker for given stock
bar(table2array(BrokerPreTrade(1,:)))
set(gca,'XTickLabel',BrokerNames.Broker)
ylabel('Market-Impact Cost (bp)')
title(['Market-Impact Cost by Broker for Symbol ' ...
    BrokerPreTrade.Properties.RowNames{1}])
grid
```



The broker `Broker8` has the highest market-impact cost and `Broker1` has the lowest one. Decide to use `Broker1` for executing the transaction using stock ABC.

For details about the preceding calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[2] Malamut, Roberto. “Multi-Period Optimization Techniques for Trade Scheduling.” Presentation at the QWAFAFEW New York Conference, April 2002.

[3] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`krg` | `marketImpact`

More About

- “Analyze Trading Execution Results” on page 6-2
- “Post-Trade Analysis Metrics Definitions” on page 6-5
- “Kissell Research Group Data Sets” on page 6-7

Optimize Trade Schedule Trading Strategy for Basket

This example shows how to optimize the strategy for a basket by minimizing trading costs using transaction cost analysis from the Kissell Research Group. Using this optimization, you determine the optimal order slicing strategy for the basket based on the trade-off between trading cost, risk, and the specified level of risk aversion. The optimization minimizes trading costs associated with the trade schedule trading strategy and a specified risk aversion parameter *Lambda*. The trading cost minimization is expressed as

$$\min[(MI + PA) + Lambda \cdot TR],$$

where trading costs are market impact *MI*, price appreciation *PA*, and timing risk *TR*.

To access the example code, enter `edit KRGTradeOptimizationExample.m` at the command line. In this example, you can run this code using a trade schedule trading strategy or a percentage of volume trading strategy. This example shows the trade schedule trading strategy. An exponential function determines the optimal trade schedule.

After executing the code in this example, you can submit an order for execution using Bloomberg, for example.

This example requires an Optimization Toolbox license. For background information, see "Optimization Theory Overview" (Optimization Toolbox).

Retrieve Market-Impact Parameters and Load Data

Retrieve the market-impact data from the Kissell Research Group FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market-impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market-impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');
close(f)

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`. Specify initial settings for the date, market-impact code, and number of trading days.

```
k = krg(miData,datetime('today'),1,250);
```

Load the example data `TradeDataTradeOpt` and the covariance data `CovarianceTradeOpt` from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox.

```
load KRGExampleData TradeDataTradeOpt CovarianceTradeOpt
```

For a description of the example data, see "Kissell Research Group Data Sets" on page 6-7.

Define Optimization Parameters

Specify initial values for risk, trading periods, portfolio value, and covariance matrix. Convert to a buy-only problem. Set the initial trade schedule.

```
% Convert table to array
CovarianceTradeOpt = table2array(CovarianceTradeOpt);
```

```
% Use total trading time of 1 day with 13 trading periods
totalDays = 1;
periodsPerDay = 13;

% Set risk aversion level
Lambda = 0.5;

% Set minimum and maximum percentage of volume
minPOV = 0.00;
maxPOV = 0.60;

% total number of trading periods
totalNumberPeriods = totalDays * periodsPerDay;

% Portfolio Value
PortfolioValue = TradeDataTradeOpt.Price'*TradeDataTradeOpt.Shares;

% Number of stocks
numberStocks = height(TradeDataTradeOpt);

% Covariance matrix is annualized covariance matrix in decimals.
% Convert to ($/Shares)^2 units for the trade period; this matrix is for a
% two-sided portfolio, buys and sells or long and short.
CC = diag(TradeDataTradeOpt.Price) * CovarianceTradeOpt * ...
    diag(TradeDataTradeOpt.Price);

% Scale to one trading period
CC = CC / periodsPerDay / k.TradeDaysInYear;

% Convert to buy-only problem (e.g., one-sided problem)
CC = TradeDataTradeOpt.SideIndicator * TradeDataTradeOpt.SideIndicator' .* CC;

% Convert Alpha_bp from basis points per day to cents/share per period
TradeDataTradeOpt.Alpha_bp = TradeDataTradeOpt.Alpha_bp / 1000 .* ...
    TradeDataTradeOpt.Price / totalNumberPeriods;

% Set the initial trade schedule or POV values
theta0 = rand(numberStocks,1);
```

Define optimization options using the `optimset` function. For details about these options, see “Optimization Options Reference” (Optimization Toolbox).

```
optionsold = optimset;
options = optimset(optionsold,'LargeScale','on','GradObj','off', ...
    'DerivativeCheck','off','FinDiffType','central','FinDiffRelStep',1E-12, ...
    'TolFun',10E-5,'TolX',10E-12,'TolCon',10E-12,'TolPCG',10E-12, ...
    'MaxFunEvals',20000,'MaxIter',20000,'DiffMinChange',10E-04);
```

Define lower and upper bounds of shares traded per interval for optimization.

```
LB = zeros(numberStocks,1);
UB = 100 * ones(numberStocks,1);
```

Minimize Trading Costs for Trade Strategy

Minimize the trading costs for the trade schedule strategy. `fmincon` finds the optimal value for the trade schedule trade strategy based on the lower and upper bound values. It does this by finding a local minimum for the trading cost. Use the objective function `optimizeTradingSchedule`. To access the code for this function, enter `edit KRGTradeOptimizationExample.m`.

```
[theta,fval,exitflag,output] = fmincon(@optimizeTradingSchedule,theta0,[], ...
    [],[],[],LB,UB,[],options,totalNumberPeriods,numberStocks,periodsPerDay, ...
    TradeDataTradeOpt,CC,Lambda,k);
```

To check whether `fmincon` found a local minimum, display the reason why the function stopped.

```
exitflag
```

```
exitflag =

     1.00
```

`fmincon` returns `1` when it finds a local minimum. For details, see `exitflag`.

Calculate shares to trade, residual shares, price appreciation, and timing risk. Then, calculate the average percentage of volume rate and trade time.

```
numPeriods = 1:totalNumberPeriods;
K_Matrix = repmat(numPeriods,numberStocks,1);
Theta_Matrix = repmat(theta,1,totalNumberPeriods);
Volume_Matrix = repmat(TradeDataTradeOpt.ADV/periodsPerDay,1, ...
    totalNumberPeriods);
TradeDataTradeOpt.VolumeProfile = Volume_Matrix;
Shares_Matrix = repmat(TradeDataTradeOpt.Shares,1,totalNumberPeriods);

% X = Shares to trade in period i
Xpct = (exp(-K_Matrix .* Theta_Matrix) .* (exp(Theta_Matrix)-1)) ./ ...
    (1 - exp(-totalNumberPeriods * Theta_Matrix));
X = repmat(TradeDataTradeOpt.Shares,1,totalNumberPeriods) .* Xpct;
TradeDataTradeOpt.TradeSchedule = X;

% R = Residual Shares at beginning of period i
Rpct = (exp(-(K_Matrix-1).*Theta_Matrix) - exp(-totalNumberPeriods.*Theta_Matrix)) ./ ...
    (1-exp(-totalNumberPeriods.*Theta_Matrix));
R = repmat(TradeDataTradeOpt.Shares,1,totalNumberPeriods) .* Rpct;

% Price Appreciations in Dollars
PA = sum(R,2) .* TradeDataTradeOpt.Alpha_bp;

% Market Impact in Dollars
MI = marketImpact(k,TradeDataTradeOpt) .* TradeDataTradeOpt.Shares .* ...
    TradeDataTradeOpt.Price ./10000;

% Timing Risk in Dollars
TR = sqrt(sum(R.^2,2) .* diag(CC));
TR_bp = TR ./ (TradeDataTradeOpt.Shares .* TradeDataTradeOpt.Price) * 10000;

% Avg POV Rate
kTR = ((TR_bp/10000*1./TradeDataTradeOpt.Volatility).^2).*(k.TradeDaysInYear*3 ./ ...
    (TradeDataTradeOpt.Shares./TradeDataTradeOpt.ADV));
POV = 1./(1+kTR);
POV = max(POV,TradeDataTradeOpt.Shares./(TradeDataTradeOpt.Shares+totalDays .* ...
    TradeDataTradeOpt.ADV));

% TradeTime
TradeDataTradeOpt.TradeTime = TradeDataTradeOpt.Shares./TradeDataTradeOpt.ADV .* ...
    (1-POV)./POV;
```

Estimate total trading costs using the optimized trade strategy.

```
TotMI = sum(MI) / (TradeDataTradeOpt.Shares' * TradeDataTradeOpt.Price) ...
    .* 10000;       % bp
TotPA = sum(PA) / (TradeDataTradeOpt.Shares' * TradeDataTradeOpt.Price) ...
    .* 10000;       % bp
TotTR = sqrt(trace(R'*CC*R)) ./ (TradeDataTradeOpt.Shares' * ...
    TradeDataTradeOpt.Price) * 10000;
```

Display total market-impact cost, price appreciation, and timing risk.

```
totalcosts = [TotMI TotPA TotTR]
```

```
totalcosts =

   38.2902         0   26.5900
```

For details about the preceding calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[2] Malamut, Roberto. “Multi-Period Optimization Techniques for Trade Scheduling.” Presentation at the QWAFAFEW New York Conference, April 2002.

[3] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`krg` | `marketImpact` | `optimset` | `fmincon` | `priceAppreciation` | `timingRisk`

More About

- “Optimize Trade Schedule Trading Strategy” on page 6-29
- “Optimize Percentage of Volume Trading Strategy” on page 6-23
- “Optimize Trade Time Trading Strategy” on page 6-26
- “Kissell Research Group Data Sets” on page 6-7

Create Basket Summary and Efficient Trading Frontier

This example shows how to evaluate trading cost and risk components for a basket using transaction cost analysis from the Kissell Research Group. To create a basket summary, estimate trading costs for the entire basket using basket optimization techniques, and then calculate risk statistics for the basket. Using the basket summary, you can provide brokers and third parties with enough information to assess the overall execution costs and trading difficulty of the basket. The basket summary enables providing transaction information without revealing the actual orders. Another way brokers use a basket summary is to assess a fair value principal bid estimate. A principal bid is a transaction where the broker charges a bid premium that is higher than the associated commission. Brokers present this transaction with guaranteed completion for a given price.

In this example, you can see a basket summary analysis table and a principal bid summary. The basket summary provides trading cost estimates for the basket across different categories, such as side, market capitalization, and market sector. The principal bid summary contains the efficient trading frontier that provides the different estimated trading costs for different time periods. The efficient trading frontier shows how cost and risk change by trading more aggressively or passively. With passive trading, market impact decreases as timing risk increases. With aggressive trading, market impact increases as timing risk decreases.

The code in this example depends on the output data from the example “Optimize Trade Schedule Trading Strategy for Basket” on page 6-60. Run the code in that example first and then run the code in this example.

To access the example code, enter `edit KRGBasketAnalysisExample.m` at the command line.

After executing the code in this example, you can submit an order for execution using Bloomberg, for example.

Estimate Trading Costs in Basket

Determine the covariance matrix. Covariance indicates how the prices of stocks in the basket relate to each other.

```
% Covariance matrix is annualized covariance matrix in decimals.
% Convert to ($/Shares)^2 units for the trade period, this matrix is for a
% two-sided portfolio, buys and sells or long and short.
diagPrice = diag(TradeDataTradeOpt.Price);
C1 = TradeDataTradeOpt.SideIndicator * TradeDataTradeOpt.SideIndicator' .* ...
    diagPrice * CovarianceTradeOpt * diagPrice;

% Covariance Matrix in $/Share^2 by Day
CD = diagPrice * CovarianceTradeOpt * diagPrice;    % compute Covariance Matrix in ($/share)^2
CD = CD / k.TradeDaysInYear;                        % scale to 1-day
CD = TradeDataTradeOpt.SideIndicator * TradeDataTradeOpt.SideIndicator' ...
    .* CD;
```

Add the estimated trading costs from the trade schedule optimization to the basket data.

```
% Market impact in basis points
TradeDataTradeOpt.MI = MI ./ (TradeDataTradeOpt.Shares .* ...
    TradeDataTradeOpt.Price) .* 10000;

% Timing risk in basis points
TradeDataTradeOpt.TR = TR ./ (TradeDataTradeOpt.Shares .* ...
    TradeDataTradeOpt.Price) .* 10000;

% Percentage of volume, price appreciation and liquidity factor
TradeDataTradeOpt.POV = POV;
```

```
TradeDataTradeOpt.PA = PA;
TradeDataTradeOpt.LF = liquidityFactor(k,TradeDataTradeOpt);
```

Calculate trading costs in basis points, cents per share, and dollars.

```
% Build optimal cost table
OptimalCostTable = table(cell(3,1),zeros(3,1),zeros(3,1),zeros(3,1), ...
    zeros(3,1),'VariableNames',{'CostUnits','MI','PA','TotalCost','TR'});
OptimalCostTable.CostUnits(1) = {'Basis Points'};
OptimalCostTable.CostUnits(2) = {'Cents per Share'};
OptimalCostTable.CostUnits(3) = {'Dollars'};

% Market impact,
OptimalCostTable.MI(1) = TotMI;
OptimalCostTable.MI(2) = TotMI / 100 * mean(TradeDataTradeOpt.Price);
OptimalCostTable.MI(3) = TotMI / 100 * (TradeDataTradeOpt.Shares' * ...
    TradeDataTradeOpt.Price);

% Price appreciation
OptimalCostTable.PA(1) = TotPA;
OptimalCostTable.PA(2) = TotPA / 100 * mean(TradeDataTradeOpt.Price);
OptimalCostTable.PA(3) = TotPA / 100 * (TradeDataTradeOpt.Shares' * ...
    TradeDataTradeOpt.Price);

% Total cost
OptimalCostTable.TotalCost(1) = TotMI + TotPA;
OptimalCostTable.TotalCost(2) = (TotMI + TotPA) / 100 * mean(TradeDataTradeOpt.Price);
OptimalCostTable.TotalCost(3) = (TotMI + TotPA) / 100 * ...
    (TradeDataTradeOpt.Shares' * TradeDataTradeOpt.Price);

% Timing risk
OptimalCostTable.TR(1) = TotTR;
OptimalCostTable.TR(2) = TotTR / 100 * mean(TradeDataTradeOpt.Price);
OptimalCostTable.TR(3) = TotTR / 100 * ...
    (TradeDataTradeOpt.Shares' * TradeDataTradeOpt.Price);
```

Display the optimal costs for the basket. Format the display output to show cents and dollars. Optimal costs are market impact, price appreciation, total cost, and timing risk.

```
format bank
OptimalCostTable
```

```
OptimalCostTable =

  3×5 table array

       CostUnits             MI          PA      TotalCost           TR
    _______________    _____________    ____    _____________    ____________

    'Basis Points'              38.30    0.00           38.30           26.57
    'Cents per Share'           14.88    0.00           14.88           10.32
    'Dollars'           171134479.73    0.00    171134479.73    118710304.48
```

Determine Risk Components in Basket

Calculate risk statistics. The marginal contribution to risk captures the risk of changing one of the components in the basket, such as the number of shares. The risk contribution is the risk for each trade in the basket.

```
% Portfolio Risk in Dollars
PortfolioRisk = sqrt(TradeDataTradeOpt.Shares' * CD * ...
    TradeDataTradeOpt.Shares);

% MCR and RC calculations
PortfolioRiskMCR = zeros(numberStocks,1);
PortfolioRiskRC =zeros(numberStocks,1);
SharesMCR = TradeDataTradeOpt.Shares;
SharesRC = TradeDataTradeOpt.Shares;
for i = 1:numberStocks
  SharesMCR(i) = TradeDataTradeOpt.Shares(i) * 0.90;
```

```
  SharesRC(i) = 0;
  PortfolioRiskMCR(i) = sqrt(SharesMCR' * CD * SharesMCR);
  PortfolioRiskRC(i) = sqrt(SharesRC' * CD * SharesRC);
end
TradeDataTradeOpt.MCR = PortfolioRisk ./ PortfolioRiskMCR - 1;
TradeDataTradeOpt.RC = PortfolioRisk ./ PortfolioRiskRC - 1;
```

Display the side, symbol, and number of shares for the safest trade in the basket using the risk contribution.

```
minrisk = min(TradeDataTradeOpt.RC);
for i = 1:25
    if TradeDataTradeOpt.RC(i) == minrisk
        idx = i;
    end
end
[TradeDataTradeOpt.Side(idx) TradeDataTradeOpt.Symbol(idx) ...
    TradeDataTradeOpt.Shares(idx)]
```

```
ans =

  1×3 cell array

    'B'    'ABC'    [100000]
```

The buy order of 100,000 shares of stock ABC contributes the most overall portfolio risk.

Create Basket Report Summary

Create a table for the basket report summary.

```
% Get sector identifiers
uniqueSectors = unique(TradeDataTradeOpt.Sector);
numSectors = size(uniqueSectors,1);
numGroups = 14 + size(uniqueSectors,1);  % Using 14 categories plus number of sectors

% Preallocate BasketReport table
BasketReport = table;
BasketReport.BasketCategory = cell(numGroups,1);
BasketReport.Number = zeros(numGroups,1);
BasketReport.Weight = zeros(numGroups,1);
BasketReport.MI = zeros(numGroups,1);
BasketReport.TR = zeros(numGroups,1);
BasketReport.POV = zeros(numGroups,1);
BasketReport.TradeTime = zeros(numGroups,1);
BasketReport.PctADV = zeros(numGroups,1);
BasketReport.Price = zeros(numGroups,1);
BasketReport.Volatility = zeros(numGroups,1);
BasketReport.Risk = zeros(numGroups,1);
BasketReport.RC = zeros(numGroups,1);
BasketReport.MCR = zeros(numGroups,1);
BasketReport.Beta = zeros(numGroups,1);
BasketReport.LF = zeros(numGroups,1);
BasketReport.TotalValue = zeros(numGroups,1);
BasketReport.BuyValue = zeros(numGroups,1);
BasketReport.SellValue = zeros(numGroups,1);
BasketReport.NetValue = zeros(numGroups,1);
BasketReport.Shares = zeros(numGroups,1);
BasketReport.BuyShares = zeros(numGroups,1);
BasketReport.SellShares = zeros(numGroups,1);
```

Calculate the basket report summary.

Divide the trades in the basket into these categories:

- `Total` — All trades in basket

- `Buy` — Buy trades
- `Cover` — Buy trades that cover a short position
- `Sell` — Sell trades
- `Short` — Short trades
- `<=1%` — Trades that have percentage of average daily volume less than or equal to 1%
- `1%-3%` — Trades that have percentage of average daily volume between 1% and 3%
- `3%-5%` — Trades that have percentage of average daily volume between 3% and 5%
- `5%-10%` — Trades that have percentage of average daily volume between 5% and 10%
- `10%-20%` — Trades that have percentage of average daily volume between 10% and 20%
- `>20%` — Trades that have percentage of average daily volume greater than 20%
- `LC` — Large-capitalization stock trades
- `MC` — Mid-capitalization stock trades
- `SC` — Small-capitalization stock trades
- `Consumer Discretionary` — Trades in the consumer discretionary industry
- `Consumer Staples` — Trades in the consumer staples industry
- `Energy` — Trades in the energy industry
- `Financials` — Trades in the financial industry
- `Health Care` — Trades in the health care industry
- `Industrials` — Trades in the industrial industry
- `Information Technology` — Trades in the information technology industry
- `Materials` — Trades in the materials industry
- `Telecommunication Services` — Trades in the telecommunication services industry
- `Utilities` — Trades in the utilities industry

For stocks in each category, calculate these values:

- `Weight` — Total trade value weight
- `MI` — Weighted average market-impact cost
- `TR` — Timing risk
- `POV` — Weighted average percentage of volume rate
- `TradeTime` — Weighted average trade time to complete the order
- `PctADV` — Weighted average order size (measured as percentage of average daily volume)
- `Price` — Weighted average share price
- `Volatility` — Weighted average volatility
- `Risk` — Portfolio risk
- `RC` — Risk contribution to the overall portfolio risk (shows the amount of risk that an order contributes to the basket)
- `MCR` — Marginal contribution to risk (shows the amount of risk that 10% of shares in the order contribute to the basket)
- `Beta` — Weighted average beta
- `LF` — Weighted average liquidity factor

- `TotalValue` — Total trade value
- `BuyValue` — Total trade value of the buy transactions
- `SellValue` — Total trade value of the sell transactions
- `NetValue` — Difference between total trade value of the buy and sell transactions
- `Shares` — Number of shares
- `BuyShares` — Number of shares to buy
- `SellShares` — Number of shares to sell

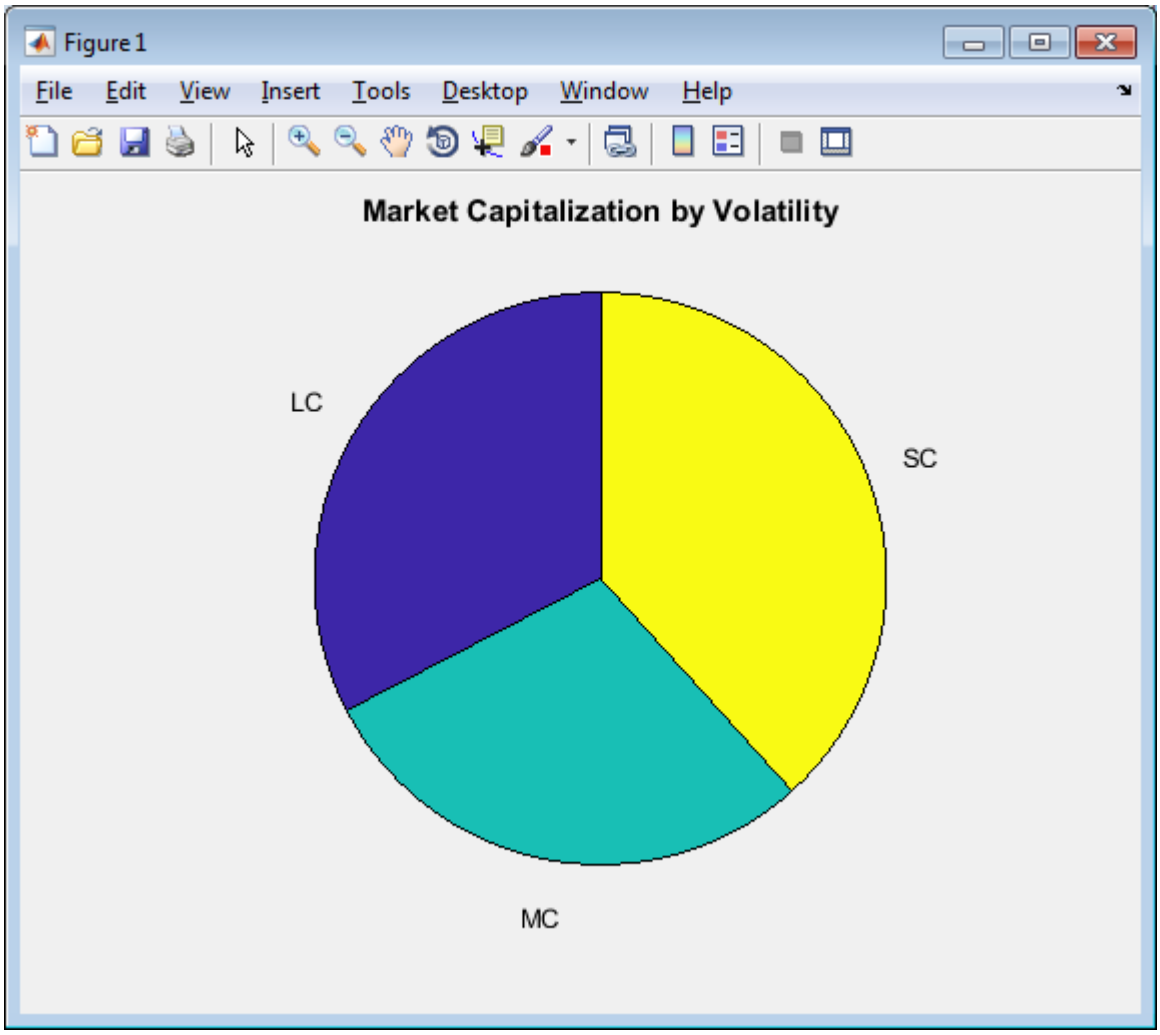
```
% Fill table, indRecord is index of matching TradeData rows
j = 0;
for i = 1:24

    switch i

      % Total
      case 1

        indRecord = true(numberStocks,1);
        BasketReport.BasketCategory(i) = {'Total'};

      % Side
      case 2
        indRecord = strcmp(TradeDataTradeOpt.Side,'B') | ...
            strcmp(TradeDataTradeOpt.Side,'Buy');
        BasketReport.BasketCategory(i) = {'Buy'};

      case 3
        indRecord = strcmp(TradeDataTradeOpt.Side,'C') | ...
            strcmp(TradeDataTradeOpt.Side,'Cover');
        BasketReport.BasketCategory(i) = {'Cover'};

      case 4
        indRecord = strcmp(TradeDataTradeOpt.Side,'S') | ...
            strcmp(TradeDataTradeOpt.Side,'Sell');
        BasketReport.BasketCategory(i) = {'Sell'};

      case 5
        indRecord = strcmp(TradeDataTradeOpt.Side,'SS') | ...
            strcmp(TradeDataTradeOpt.Side,'Short') | ...
            strcmp(TradeDataTradeOpt.Side,'Sell Short');
        BasketReport.BasketCategory(i) = {'Short'};

      % Liquidity Category
      case 6

        % Percentage of average daily volume is less than 1 %
        indRecord = (TradeDataTradeOpt.PctADV <= 0.01);
        BasketReport.BasketCategory(i) = {'<=1%'};

      case 7

        % Percentage of average daily volume is between 1 and 3 %
        indRecord = (TradeDataTradeOpt.PctADV > 0.01 & ...
            TradeDataTradeOpt.PctADV <= 0.03);
        BasketReport.BasketCategory(i) = {'1%-3%'};

      case 8

        % Percentage of average daily volume is between 3 and 5 %
        indRecord = (TradeDataTradeOpt.PctADV > 0.03 & ...
            TradeDataTradeOpt.PctADV <= 0.05);
        BasketReport.BasketCategory(i) = {'3%-5%'};

      case 9

        % Percentage of average daily volume is between 5 and 10 %
        indRecord = (TradeDataTradeOpt.PctADV > 0.05 & ...
            TradeDataTradeOpt.PctADV <= 0.10);
        BasketReport.BasketCategory(i) = {'5%-10%'};

      case 10

        % Percentage of average daily volume is between 10 and 20 %
        indRecord = (TradeDataTradeOpt.PctADV > 0.10 & ...
```

```
        TradeDataTradeOpt.PctADV <= 0.20);
    BasketReport.BasketCategory(i) = {'10%-20%'};

  case 11

    % Percentage of average daily volume is greater than 20 %
    indRecord = (TradeDataTradeOpt.PctADV > 0.20);
    BasketReport.BasketCategory(i) = {'>20%'};

  % Market cap
  case 12

    % Large cap
    indRecord = (TradeDataTradeOpt.MktCap > 10000000000);
    BasketReport.BasketCategory(i) = {'LC'};

  case 13

    % Mid cap
    indRecord = (TradeDataTradeOpt.MktCap > 1000000000 & ...
        TradeDataTradeOpt.MktCap <= 10000000000);
    BasketReport.BasketCategory(i) = {'MC'};

  case 14

    % Small cap
    indRecord = (TradeDataTradeOpt.MktCap <= 1000000000);
    BasketReport.BasketCategory(i)={'SC'};

  % Sectors
  % Description of basket category
  case {15, 16, 17, 18, 19, 20, 21, 22, 23, 24}
    j = j + 1;
    if j <= numSectors
      indRecord = strcmp(TradeDataTradeOpt.Sector,uniqueSectors(j));
      BasketReport.BasketCategory(i) = uniqueSectors(j);
    end

end

% Get subset of TradeData
TD = TradeDataTradeOpt(indRecord,:);

if ~isempty(TD)

    % Covariance Matrix in $/Shares^2
  CC2 = CC(indRecord,indRecord);    %Trading Period Covariance Matrix in $/Shares^2
  C2 = C1(indRecord,indRecord);     %Annualized Covariance Matrix in $/Shares^2
  RR = R(indRecord,:);              %Residuals for Stocks in group

  % Basket Summary Calculations
  Weight2 = TD.Value / sum(TD.Value);

  % Side
  I_Buy = (TD.SideIndicator == 1);
  I_Sell = (TD.SideIndicator == -1);

  % Fill basket report table
  BasketReport.Number(i) = size(TD,1);                    % Number of records that match criteria
  BasketReport.Weight(i) = sum(TD.Value)/PortfolioValue;  % Weight of assets in criteria
  BasketReport.MI(i) = Weight2' * TD.MI;                  % Market impact of assets
  BasketReport.TR(i) = sqrt(trace(RR'*CC2*RR)) / sum(TD.Value) * 10000;   % Timing risk of assets
  BasketReport.POV(i) = Weight2' * TD.POV;                % POV of assets
  BasketReport.TradeTime(i) = Weight2' * TD.TradeTime;    % Tradetime of assets
  BasketReport.PctADV(i) = Weight2' * TD.PctADV;          % Percentage of ADV
  BasketReport.Price(i) = Weight2' * TD.Price;            % Total price of assets
  BasketReport.Volatility(i) = Weight2' * TD.Volatility;  % Volatility
  BasketReport.Risk(i) = sqrt(TD.Shares' * C2 * TD.Shares) / ...
      sum(TD.Value);   % Risk value

  % RC and MCR
  Shares2 = TradeDataTradeOpt.Shares;
  Shares3 = TradeDataTradeOpt.Shares;
  Shares2(indRecord) = 0;
  Shares3(indRecord) = Shares3(indRecord) * 0.90;

  if sum(Shares2) > 0
    BasketReport.RC(i) = PortfolioRisk / sqrt(Shares2' * CD * Shares2) - 1;
  else
    BasketReport.RC(i) = 0;
  end
```

```
        BasketReport.MCR(i) = PortfolioRisk / sqrt(Shares3' * CD * Shares3) - 1;

        % Beta value, liquidity factor and total value
        BasketReport.Beta(i) = sum(Weight2 .* TD.SideIndicator .* TD.Beta);
        BasketReport.LF(i) = Weight2' * TD.LF;
        BasketReport.TotalValue(i) = sum(TD.Value);

        % Calculate buy share values
        if sum(I_Buy) > 0
          BasketReport.BuyValue(i) = sum(TD.Value(I_Buy));
          BasketReport.BuyShares(i) = sum(TD.Shares(I_Buy));
        else
          BasketReport.BuyValue(i) = 0;
          BasketReport.BuyShares(i) = 0;
        end

        % Calculate sell share values
        if sum(I_Sell) > 0
          BasketReport.SellValue(i) = sum(TD.Value(I_Sell));
          BasketReport.SellShares(i) = sum(TD.Shares(I_Sell));
        else
          BasketReport.SellValue(i) = 0;
          BasketReport.SellShares(i) = 0;
        end

        % Calculate net value of criteria and number of shares
        BasketReport.NetValue(i) = BasketReport.BuyValue(i) - ...
            BasketReport.SellValue(i);
        BasketReport.Shares(i) = sum(TD.Shares);

    end
end

% Remove rows with no stocks
indRecord = (BasketReport.Number > 0);
BasketReport = BasketReport(indRecord,:);
```

Display market capitalization by volatility as a pie chart.

```
pie(BasketReport.Volatility(8:10),BasketReport.BasketCategory(8:10))
title('Market Capitalization by Volatility')
```



Create Principal Bid Summary

Determine the efficient trading frontier by time. Use different trade time scenarios. Estimate trading costs for price appreciation, market impact, and timing risk for each scenario.

```
ScenarioTime = [0.10;0.25;0.50;0.75;1.0;1.50;2.0;2.5;3.0;3.5;4.0;4.5;5.0];
numScenarios = size(ScenarioTime,1);
ETFCosts = zeros(numScenarios,5);

TableVariableNames = TradeDataTradeOpt.Properties.VariableNames;
if sum(strcmp(TableVariableNames,'DeltaP')) > 0
    DeltaP = TradeDataTradeOpt.DeltaP;
elseif sum(strcmp(TableVariableNames,'Alpha_bp')) > 0
    DeltaP = TradeDataTradeOpt.Alpha_bp;
else
    DeltaP = zeros(NumberStocks,1);
end

% Convert DeltaP from basis points per day to cents/share per period
DeltaP = DeltaP / 1000 .* TradeDataTradeOpt.Price / totalNumberPeriods;

for i = 1:numScenarios

    TradeTime = ScenarioTime(i);
    TradeDataTradeOpt.POV = TradeDataTradeOpt.Shares ./ ...
        (TradeDataTradeOpt.Shares + TradeTime .* TradeDataTradeOpt.ADV);

    % Price Appreciations in Dollars
```

```
    PA = 1/2 * TradeDataTradeOpt.Shares .* DeltaP .* TradeTime;
    TotPA = sum(PA) / (TradeDataTradeOpt.Shares' * ...
        TradeDataTradeOpt.Price) .* 10000;          % bp
    PA = PA ./ (TradeDataTradeOpt.Shares .* ...
        TradeDataTradeOpt.Price) * 10000;           % bp

    % Market Impact in Dollars
    MI = marketImpact(k,TradeDataTradeOpt) .* TradeDataTradeOpt.Shares .* ...
        TradeDataTradeOpt.Price ./ 10000; %dollars;
    TotMI = sum(MI) / (TradeDataTradeOpt.Shares' * ...
        TradeDataTradeOpt.Price) .* 10000;          % bp
    MI = MI ./ (TradeDataTradeOpt.Shares .* ...
        TradeDataTradeOpt.Price) * 10000;           % bp

    % Timing Risk in Dollars
    TotTR = sqrt(1/3 * TradeDataTradeOpt.Shares' * ...
        (CD * TradeTime) * TradeDataTradeOpt.Shares) / ...
        (TradeDataTradeOpt.Shares' * TradeDataTradeOpt.Price) * 10000;

    % Total Cost Dollars
    TotTC = (TotMI + TotPA);

    % ETF Cost Table
    ETFCosts(i,1) = TradeTime;
    ETFCosts(i,2) = TotMI;
    ETFCosts(i,3) = TotPA;
    ETFCosts(i,4) = TotTC;
    ETFCosts(i,5) = TotTR;

end

% Save as Table
ETFCosts = table(ETFCosts(:,1),ETFCosts(:,2),ETFCosts(:,3),ETFCosts(:,4), ...
    ETFCosts(:,5),'VariableNames',{'Days','MI_bp','PA_bp','TotalCost_bp', ...
    'TR_bp'});
```

Determine the trade time with the lowest total cost.

```
mintotcost = min(ETFCosts.TotalCost_bp);
for i = 1:numScenarios
    if(ETFCosts.TotalCost_bp(i) == mintotcost)
        scenario = ETFCosts.Days(i);
    end
end
scenario
```

```
scenario =

     5
```

For details about the preceding calculations, contact the Kissell Research Group.

References

[1] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[2] Malamut, Roberto. “Multi-Period Optimization Techniques for Trade Scheduling.” Presentation at the QWAFAFEW New York Conference, April 2002.

[3] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`krg` | `liquidityFactor` | `marketImpact`

More About

- “Optimize Trade Schedule Trading Strategy for Basket” on page 6-60
- “Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17
- “Liquidate Dollar Value from Portfolio” on page 6-43
- “Kissell Research Group Data Sets” on page 6-7

Money.Net Web Socket Interface Topics

- “Retrieve Current and Historical Money.Net Data Using Money.Net Web Socket Interface” on page 7-2
- “Retrieve Real-Time Money.Net Data Using Money.Net Web Socket Interface” on page 7-4
- “Retrieve Money.Net News Stories Using Money.Net Web Socket Interface” on page 7-6

Retrieve Current and Historical Money.Net Data Using Money.Net Web Socket Interface

This example shows how to retrieve current data for symbols, historical data, and current data for option symbols from Money.Net using the Money.Net web socket interface.

To run this example, you need a Money.Net user name and password. To request these credentials, contact Money.Net.

Create Money.Net Connection

Create a Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve Money.Net Current Data

Retrieve Money.Net current data `d` for the symbol IBM using the Money.Net connection `c`. The table `d` contains the variables for all Money.Net fields.

```
symbol = "IBM";
d = getdata(c,symbol);
```

Retrieve Money.Net current data for the `symbols` list that contains the IBM and Google® symbols. Specify the Money.Net data fields `f` for highest and lowest prices for the current trading day.

```
symbols = ["IBM" "GOOG"];
f = ["high" "low"];
d = getdata(c,symbols,f);
```

`d` is a table that contains the variables for high and low prices. The rows contain Money.Net data values for each symbol in the symbol list.

Retrieve Money.Net Historical Data

Retrieve historical data in daily bars for the symbol IBM. Specify the date range from June 1, 2015, through June 5, 2015, using `datetime`. To retrieve daily data, specify the interval as `"1D"`. Retrieve only the high and low price fields `f` from Money.Net.

`d` is a timetable that contains these variables:

- Date timestamp
- High price
- Low price

```
s = "IBM";
date = [datetime("1-Jun-2015") datetime("5-Jun-2015")];
interval = "1D";
f = ["high" "low"];
d = timeseries(c,s,date,interval,f);
```

Determine the average high price in the date range.

```
avghigh = mean(d.high);
```

Retrieve Money.Net Option Symbol Data

Retrieve option symbols `o` for the symbol IBM. `o` is a table. Each row of the table is an option symbol.

```
s = "IBM";
o = optionchain(c,s);
```

Retrieve the current data for the first option symbol `o.symbol(1)`. Specify fields `f` for describing the option symbol:

- Option symbol description
- Option symbol ask price

`d` is a table with one row of data. The row name is the option symbol name. The table contains a variable for each specified field `f`.

```
symbol = o.symbol(1);
f = ["description" "askPrice"];
d = getdata(c,symbol,f);
```

Close Money.Net Connection

```
close(c)
```

See Also

`moneynetws` | `isconnection` | `getdata` | `optionchain` | `timeseries` | `close`

More About

- “Retrieve Real-Time Money.Net Data Using Money.Net Web Socket Interface” on page 7-4
- “Retrieve Money.Net News Stories Using Money.Net Web Socket Interface” on page 7-6

External Websites

- Money.Net Help

Retrieve Real-Time Money.Net Data Using Money.Net Web Socket Interface

This example shows how to retrieve real-time data from Money.Net for a symbol. It explains how to subscribe to real-time updates, stop subscription, and process real-time updates using a custom event handler function. The example uses the Money.Net web socket interface to create a Money.Net connection.

To process real-time data updates, you can use the default event handler function. For a different approach, you can write a custom event handler function. To write custom event handler functions with Money.Net data, see `realtime`. For custom event handler functions, see “Writing and Running Custom Event Handler Functions” on page 1-26.

This example requires a Money.Net user name and password. To request these credentials, contact Money.Net.

Create Money.Net Connection

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve Real-Time Data for One Symbol

Retrieve Money.Net real-time data updates for the IBM symbol.

```
symbol = "IBM";
realtime(c,symbol)
```

The default event handler `mnWSRealTimeEventHandler` processes all real-time data updates. To access the code for the default event handler, enter `edit mnWSRealTimeEventHandler.m`.

The `mnWSRealTimeEventHandler` function creates the workspace variable `IBMRealTime`. The `mnWSRealTimeEventHandler` function populates the table `IBMRealTime` with real-time data updates. To see the real-time data, open `IBMRealTime` in the Variables editor.

Stop Real-Time Data Updates

Stop the symbol subscription.

```
stop(c)
```

`mnWSRealTimeEventHandler` stops processing all real-time data updates. The last real-time data update remains in `IBMRealTime`.

Retrieve Real-Time Data Using Custom Event Handler Function

Define a custom event handler function `myfcn`. The `myfcn` function displays Money.Net real-time data to the Command Window.

```
myfcn = @(x)disp(x);
```

Retrieve Money.Net real-time data updates for the IBM symbol using `myfcn`.

```
symbol = "IBM";
realtime(c,symbol,myfcn)
```

`myfcn` displays real-time data updates for IBM in the Command Window.

Stop the symbol subscription.

```
stop(c,symbol)
```

`myfcn` stops displaying real-time data updates in the Command Window.

Close Money.Net Connection

```
close(c)
```

See Also

`moneynetws` | `isconnection` | `getsubscriptions` | `realtime` | `stop` | `close`

More About

- “Writing and Running Custom Event Handler Functions” on page 1-26
- “Retrieve Current and Historical Money.Net Data Using Money.Net Web Socket Interface” on page 7-2
- “Retrieve Money.Net News Stories Using Money.Net Web Socket Interface” on page 7-6

External Websites

- Money.Net Help

Retrieve Money.Net News Stories Using Money.Net Web Socket Interface

This example shows how to retrieve news stories from Money.Net in different ways using the Money.Net web socket interface. You can search for a specific number of news stories, search for news stories using specific filter criteria, or you can stream news stories in real time.

To run this example, you need a Money.Net user name and password. To request these credentials, contact Money.Net.

Create Money.Net Connection

Create the Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve Specific Number of News Stories

Retrieve news data `n` for 10 news stories using the Money.Net connection `c`.

```
n = news(c,Number=10);
```

Search News Stories Using Search Term

Retrieve news stories that mention the term Windows. `n` is a table with data for 50 news stories.

```
term = "Windows";
n = news(c,SearchTerm=term);
```

Search News Stories Using Category

Retrieve news stories in the general finance category. `n` is a table with data for 50 news stories.

```
category = "General Finance";
n = news(c,Category=category);
```

Search News Stories Using Symbol

Retrieve news stories that contain the symbol for Microsoft. `n` is a table with data for 50 news stories.

```
symbol = "MSFT";
n = news(c,Symbol=symbol);
```

Analyze News Stories for Analyst Ratings

Search the analyst ratings category for Microsoft. Return 100 news stories.

```
symbol = "MSFT";
category = "Analyst Ratings";
n = news(c,Number=100,Symbol=symbol,Category=category);
```

Perform a non-case-sensitive search of the headlines using `contains`. Here, assume that the word `"buy"` represents a buy rating for Microsoft from an investment analyst. Count the occurrences of buy ratings in the 100 news stories.

```
headlines = n.headline;
sentiment = contains(headlines,"buy",IgnoreCase=true);
buys = sum(sentiment);
```

To compare buy ratings against sell and hold ratings, replace `"buy"` with the corresponding term and count the occurrences. With these counts, you can see which ratings are more common.

Stream News Stories in Real Time

Start the subscription to the Money.Net real-time news data stream using the default event handler function `mnNewsStreamEventHandler`. The function `mnNewsStreamEventHandler` processes news data events by populating the workspace variable `mnNewsStreamLatest` with the latest news stories. News stories populate in the `mnNewsStreamLatest` variable until it contains 10 rows. Then, the latest news stories overwrite the older ones in `mnNewsStreamLatest`. To access the code for this function, enter `edit mnNewsStreamEventHandler.m`.

```
news(c,Subscription="on")
```

The workspace variable `mnNewsStreamLatest` appears in the MATLAB Workspace. To see the latest 10 news stories, open `mnNewsStreamLatest` in the Variables editor.

Stop the real-time news data stream.

```
news(c,Subscription="off")
```

Money.Net stops updating news stories in `mnNewsStreamLatest`.

Close Money.Net Connection

```
close(c)
```

See Also

`moneynetws` | `isconnection` | `news` | `close` | `contains`

More About

- “Retrieve Current and Historical Money.Net Data Using Money.Net Web Socket Interface” on page 7-2
- “Retrieve Real-Time Money.Net Data Using Money.Net Web Socket Interface” on page 7-4

External Websites

- Money.Net Help

Money.Net Topics

Retrieve Current and Historical Money.Net Data

This example shows how to retrieve current data for symbols, historical data, and current data for option symbols from Money.Net.

To run this example, you need a Money.Net user name and password. To request these credentials, contact Money.Net.

To access the code for this example, enter `edit MoneyNetDataWorkflowExample.m`.

Create Money.Net Connection

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve Money.Net Current Data

Retrieve Money.Net current data `d` for the symbol IBM using the Money.Net connection `c`. Specify the Money.Net data fields `f` for ask and bid price.

```
symbol = 'IBM';
f = {'Ask','Bid'};

d = getdata(c,symbol,f);
```

Display Money.Net current data. `d` is a table that contains the variables for symbol, ask price, and bid price. The row contains Money.Net data values for each variable.

```
d
```

```
d =

    Symbol     Ask       Bid
    ______    ______    ______

    'IBM'     145.00    143.85
```

Retrieve Money.Net current data for the `symbols` list that contains these symbols: IBM, Google, and Yahoo!®.

```
symbols = {'IBM','GOOG','YHOO'};

d = getdata(c,symbols,f);
```

Display Money.Net current data. `d` is a table that contains the variables for symbol, ask price, and bid price. The rows contain Money.Net data values for each symbol in the symbol list.

```
d
```

```
d =

    Symbol     Ask       Bid
    ______    ______    ______
```

```
    'IBM'     145.00    143.85
    'GOOG'    700.50    700.05
    'YHOO'     37.50     37.41
```

Retrieve Money.Net Historical Data

Retrieve historical data in daily bars for the symbol IBM. Specify the date range from June 1, 2015, through June 5, 2015, using `datetime`. To retrieve daily data, specify the interval as `'1D'`. Retrieve only the high and low price fields `f` from Money.Net.

`d` is a table that contains these variables:

- Date timestamp
- High price
- Low price

```
s = 'IBM';
date = [datetime('1-Jun-2015') datetime('5-Jun-2015')];
interval = '1D';
f = {'High','Low'};

d = timeseries(c,s,date,interval,f);
```

Display the first three rows of daily data `d`.

```
d(1:3,:)
```

```
ans =

          Date             High      Low
    ___________________    ______    ______

    06/01/15 00:00:00    171.04    169.03
    06/02/15 00:00:00    170.45    168.43
    06/03/15 00:00:00    171.56    169.63
```

Determine the average high price in the date range.

```
mean(d.High)
```

```
ans =

        170.51
```

Retrieve Money.Net Option Symbol Data

Retrieve option symbols `o` for the symbol IBM. `o` is a cell array of character vectors. Each character vector is an option symbol.

```
s = 'IBM';

o = optionchain(c,s);
```

Display the first three option symbols.

```
o(1:3)
```

```
ans =
```

```
3×1 cell array

  'O:IBM\16F24\131 .0'
  'O:IBM\16R24\135 .0'
  'O:IBM\16F24\142 .0'
```

Retrieve the current data for the first option symbol `o(1)` and display it. Specify fields `f` for describing the option symbol:

- Option symbol description
- Option symbol strike
- Option symbol expiration date
- Option symbol ask price
- Option symbol bid price

`d` is a table with one row of data. The data contains the option symbol name in the first variable and a variable for each specified field `f`.

```
symbol = o(1);
f = {'Description','Strike','Expiration','Ask','Bid'};

d = getdata(c,symbol,f)
```

```
d =

          Symbol                Description             Strike    Expiration     Ask      Bid
    ____________________    ____________________________    ______    __________    _____    _____

    'O:IBM\16F24\131 .0'    'IBM Call 06/24/2016 131.0'    131       06/24/16      23.75    21.75
```

Close Money.Net Connection

```
close(c)
```

See Also

`moneynet` | `getdata` | `timeseries` | `close` | `optionchain`

More About

- “Retrieve Real-Time Money.Net Data” on page 8-5
- “Retrieve Money.Net News Stories” on page 8-7
- “Money.Net Error and Warning Messages” on page 8-10

External Websites

- Money.Net API Documentation
- Money.Net Help

Retrieve Real-Time Money.Net Data

This example shows how to retrieve real-time data from Money.Net for a symbol. It explains how to subscribe to real-time updates, stop subscription, and process real-time updates using a custom event handler function.

To process real-time data updates, you can use the default event handler function. Or, for a different approach, you can write a custom event handler function. For writing custom event handler functions with Money.Net data, see `realtime`. For custom event handler functions, see “Writing and Running Custom Event Handler Functions” on page 1-26.

This example requires a Money.Net user name and password. To request these credentials, contact Money.Net.

To access the code for this example, enter `edit MoneyNetDataWorkflowExample.m`.

Create Money.Net Connection

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve Real-Time Data for One Symbol

Retrieve Money.Net real-time data updates for the IBM symbol.

```
symbol = 'IBM';

realtime(c,symbol)
```

The default event handler `mnRealTimeEventHandler` processes all real-time data updates. To access the code for the default event handler, enter `edit mnRealTimeEventHandler.m`.

The `mnRealTimeEventHandler` function creates the workspace variable `IBMRealTime`. The `mnRealTimeEventHandler` function populates the table `IBMRealTime` with real-time data updates. To see the real-time data, open `IBMRealTime` in the Variables editor.

Stop Real-Time Data Updates

Stop the symbol subscription.

```
stop(c)
```

`mnRealTimeEventHandler` stops processing all real-time data updates. The last real-time data update remains in `IBMRealTime`.

Retrieve Real-Time Data Using Custom Event Handler Function

Define a custom event handler function `myfcn`. The `myfcn` function displays Money.Net real-time data to the Command Window.

```
myfcn = @(x)disp(x);
```

Retrieve Money.Net real-time data updates for the IBM symbol using `myfcn`.

```
symbol = 'IBM';

realtime(c,symbol,myfcn)
```

Symbol	Description	Yesterday	YesterdayDateTime	Bid	Ask	ExchangeOfTheCurrentBidPrice	Ex
'IBM'	'INTERNATIONAL BUSINESS MACHS'	148.31	05/24/16 00:00:00	151.65	151.67	''	''

`myfcn` displays real-time data updates for IBM in the Command Window.

Stop the symbol subscription.

```
stop(c,symbol)
```

`myfcn` stops displaying real-time data updates in the Command Window.

Close Money.Net Connection

```
close(c)
```

See Also

`moneynet` | `realtime` | `stop` | `close`

More About

- “Retrieve Current and Historical Money.Net Data” on page 8-2
- “Retrieve Money.Net News Stories” on page 8-7
- “Writing and Running Custom Event Handler Functions” on page 1-26
- “Money.Net Error and Warning Messages” on page 8-10

External Websites

- Money.Net API Documentation
- Money.Net Help

Retrieve Money.Net News Stories

This example shows how to retrieve news stories from Money.Net in different ways. You can search for a specific number of news stories. You can search for news stories using specific filter criteria. Or, you can stream news stories in real time.

To run this example, you need a Money.Net user name and password. To request these credentials, contact Money.Net.

To access the code for this example, enter `edit MoneyNetNewsWorkflowExample.m`.

Create Money.Net Connection

Create the Money.Net connection `c` using the user name `username` and password `pwd`.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve Specific Number of News Stories

Retrieve news data `n` for 10 news stories using the Money.Net connection `c`.

```
n = news(c,'Number',10);
```

Display the news story title, identifier, and published time for the first news story in the table `n`.

```
n(1,1:3)
```

```
ans =
                            ArticleTitle                              ArticleID       PublishedTime
    ______________________________________________________________    _________    ________________

    'Stop talking about replacements. Give PC owners something new al…'    3.8917e+09    05/13/16 10:00:02
```

Search News Stories Using Search Term

Retrieve news stories that mention the term Windows. `n` is a table with data for 50 news stories.

```
term = 'Windows';

n = news(c,'SearchTerm',term);
```

Display the news story title, identifier, and published time for the first news story.

```
n(1,1:3)
```

```
ans =
                            ArticleTitle                              ArticleID       PublishedTime
    ______________________________________________________________    _________    ________________

    'LogMein Shares Edge Lower; LastPass Says Browser Extension Now A…'    4.0005e+09    06/08/16 13:22:04
```

Search News Stories Using Category

Retrieve news stories in the general finance category. `n` is a table with data for 50 news stories.

```
category = 'General Finance';

n = news(c,'Category',category);
```

Display the news story title, identifier, and published time for the first news story.

```
n(1,1:3)
```

```
ans =
                              ArticleTitle                                ArticleID       PublishedTime
    ______________________________________________________________________    _________    ________________

    'Keep calm and ooze compassion: Leave must seize the moral high g…'    4.0007e+09    06/08/16 12:48:42
```

Search News Stories Using Symbol

Retrieve news stories that contain the symbol for Microsoft. `n` is a table with data for 50 news stories.

```
symbol = 'MSFT';

n = news(c,'Symbol',symbol);
```

Display the news story title, identifier, and published time for the first news story.

```
n(1,1:3)
```

```
ans =
                    ArticleTitle                       ArticleID       PublishedTime
    ___________________________________________________    _________    ________________

    'Microsoft announces after party to Apple's WWDC'    4.0005e+09    06/08/16 12:51:49
```

Analyze News Stories for Analyst Ratings

Search the analyst ratings category for Microsoft. Return 100 news stories.

```
symbol = 'MSFT';
category = 'Analyst Ratings';

n = news(c,'Number',100,'Symbol',symbol,'Category',category);
```

Convert news story titles to the string array `titles`.

```
titles = string(n.ArticleTitle);
```

Perform a non-case-sensitive search of the titles using `contains`. Here, assume that the word `'buy'` represents a buy rating for Microsoft from an investment analyst. Count the occurrences of buy ratings in the 100 news stories.

```
sentiment = contains(titles,'buy','IgnoreCase',true);

sum(sentiment)

ans =

    33
```

To compare buy ratings against sell and hold ratings, replace `'buy'` with the corresponding term and count the occurrences. With these counts, you can see which ratings are more common.

Stream News Stories in Real Time

Start the subscription to the Money.Net real-time news data stream using the default event handler function `mnNewsStreamEventHandler`. The function `mnNewsStreamEventHandler` processes news data events by populating the workspace variable `mnNewsStreamLatest` with the latest news stories. News stories populate in the `mnNewsStreamLatest` variable until it contains 10 rows. Then, the latest news stories overwrite the older ones in `mnNewsStreamLatest`. To access the code for this function, enter `edit mnNewsStreamEventHandler.m`.

```
news(c,'Subscription','on')
```

The workspace variable `mnNewsStreamLatest` appears in the MATLAB Workspace.

Display the news story title, identifier, and published time for the first news story.

```
mnNewsStreamLatest(1,1:3)
```

```
ans =
                            ArticleTitle                                ArticleID       PublishedTime
    ____________________________________________________________________    _________    ________________

    'Stop talking about replacements. Give PC owners something new al…'    3.8917e+09    05/13/16 10:00:02
```

To see the latest 10 news stories, open `mnNewsStreamLatest` in the Variables editor.

Stop the real-time news data stream.

```
news(c,'Subscription','off')
```

Money.Net stops updating news stories in `mnNewsStreamLatest`.

Close Money.Net Connection

```
close(c)
```

See Also

`moneynet` | `close` | `news` | `contains`

More About

- “Retrieve Current and Historical Money.Net Data” on page 8-2
- “Retrieve Real-Time Money.Net Data” on page 8-5
- “Money.Net Error and Warning Messages” on page 8-10

External Websites

- Money.Net API Documentation
- Money.Net Help

Money.Net Error and Warning Messages

Address any error or warning messages that you encounter while connecting to or retrieving data from Money.Net using these tables.

Money.Net Connection Error Messages

Connection Error Message	Probable Causes	Resolution
Failed to connect to the Money.Net data servers. Confirm that the port number is valid.	The specified port number is invalid. A firewall or proxy is denying the connection.	To specify the default port number, use the first syntax in `moneynet`. To set up a firewall or proxy to work with Money.Net, see Money.Net Firewall Instructions. Your firewall must support a direct internet connection to Money.Net. For details, see Money.Net Firewall Instructions.
Invalid username or password.	The user name and password combination is invalid.	Verify your user name and password. To validate these credentials, contact Money.Net.
User *`username`* is already logged into Money.Net.	A Money.Net connection is open with the specified user name. Only one connection can exist for each user name at a time.	First, to close the previous Money.Net connection, run the `close` function. Then, to create a Money.Net connection, run the `moneynet` function.
Lost connection to the Money.Net data server. Use the MONEYNET command to connect again.	The Money.Net data server interrupts the connection. You create multiple `moneynet` objects.	To create a Money.Net connection, run the `moneynet` function.
MONEYNET object is not connected to the Money.Net data servers. Use MONEYNET to create a new connection.	You execute a function using a closed Money.Net connection.	To create a Money.Net connection, run the `moneynet` function.

Money.Net Data Retrieval Error Messages

Data Retrieval Error Message	Probable Causes	Resolution
Unrecognized data field *`Field`*.	The Money.Net field for the field input argument in the `getdata` or `timeseries` functions is invalid.	Verify the Money.Net field name. To view the list of valid Money.Net fields and field definitions, see the Money.Net API Documentation.

Data Retrieval Error Message	Probable Causes	Resolution
Unrecognized historical field *`Field`*.	The Money.Net field for the field input argument in the `getdata` or `timeseries` functions is invalid.	Verify the Money.Net field name. To view the list of valid Money.Net fields and field definitions, see the Money.Net API Documentation.
Invalid combination of Name-Value pairs. Type HELP MONEYNET/NEWS to see the valid syntax	The combination of name-value pair arguments in the `news` function is invalid.	Verify the name-value pair argument syntax when running `news`.

Money.Net Data Retrieval Warning Messages

Data Retrieval Warning Message	Probable Causes	Resolution
No timeseries data returned.	No intraday or historical data that matches the specified input arguments in the `timeseries` function is available. The specified historical date range is outside of the available dates from Money.Net. Second and minute interval data is only available for more recent dates.	Verify that the symbol and interval input arguments are valid. For details, see `timeseries`. Verify that the specified dates contain activity. For example, no data is available on weekends or holidays. Specify a recent date range.
No option month data returned for *`Symbol`*.	The specified symbol is invalid. No options are available for the specified symbol.	Verify the symbol is valid.
No news stories found that match the search criteria.	No news stories are available that match the specified criteria in `news`.	Change the search criteria for news stories. For details, see `news`.
Error occurred while processing real-time data:	The custom event handler function returns an error when processing real-time data.	To find the cause of the error, troubleshoot the custom event handler function code. For writing a custom event handler function using Money.Net data, see `realtime`. For custom event handlers, see “Writing and Running Custom Event Handler Functions” on page 1-26.

Data Retrieval Warning Message	Probable Causes	Resolution
Error occurred while processing real-time news:	The custom event handler function returns an error when processing real-time news data.	To find the cause of the error, troubleshoot the custom event handler function code. For writing a custom event handler function using Money.Net data, see `realtime`. For custom event handlers, see “Writing and Running Custom Event Handler Functions” on page 1-26.

See Also

`moneynet` | `realtime` | `close` | `news` | `timeseries`

More About

- “Retrieve Current and Historical Money.Net Data” on page 8-2
- “Retrieve Real-Time Money.Net Data” on page 8-5
- “Retrieve Money.Net News Stories” on page 8-7
- “Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

- Money.Net API Documentation
- Money.Net Help

Twitter Topics

Conduct Sentiment Analysis Using Historical Tweets

This example shows how to search and retrieve all available Tweets in the last 7 days and import them into MATLAB. After importing the data, you can conduct sentiment analysis. This analysis enables you to determine subjective information, such as moods, opinions, or emotional reactions, from text data. This example searches for positive and negative moods regarding the financial services industry.

To run this example, you need Twitter credentials. To obtain these credentials, you must first log in to your Twitter account. Then, fill out the form in Create an application.

To access the example code, enter `edit TwitterExample.m` at the command line.

Connect to Twitter

Create a Twitter connection using your credentials. (The values in this example do not represent real Twitter credentials.)

```
consumerkey = 'abcdefghijklmnop123456789';
consumersecret = 'qrstuvwxyz123456789';
accesstoken = '123456789abcdefghijklmnop';
accesstokensecret = '123456789qrstuvwxyz';

c = twitter(consumerkey,consumersecret,accesstoken,accesstokensecret);
```

Check the Twitter connection. If the `StatusCode` property has the value `OK`, the connection is successful.

```
c.StatusCode

ans =

    OK
```

Retrieve Latest Tweets

Search for the latest 100 Tweets about the financial services industry using the Twitter connection object. Use the search term `financial services`. Import Tweet® data into the MATLAB workspace.

```
tweetquery = 'financial services';
s = search(c,tweetquery,'count',100);
statuses = s.Body.Data.statuses;
pause(2)
```

`statuses` contains the Tweet data as a cell array of 100 structures. Each structure contains a field for the Tweet text, and the remaining fields contain other information about the Tweet.

Search and retrieve the next 100 Tweets that have occurred since the previous request.

```
sRefresh = search(c,tweetquery,'count',100, ...
    'since_id',s.Body.Data.search_metadata.max_id_str);
statuses = [statuses;sRefresh.Body.Data.statuses];
```

`statuses` contains the latest 100 Tweets in addition to the previous 100 Tweets.

Retrieve All Available Tweets

Retrieve all available Tweets about the financial services industry using a `while` loop. Check for available data using the `isfield` function and the structure field `next_results`.

```
while isfield(s.Body.Data.search_metadata,'next_results')
    % Convert results to string
    nextresults = string(s.Body.Data.search_metadata.next_results);
    % Extract maximum Tweet identifier
    max_id = extractBetween(nextresults,"max_id=","&");
    % Convert maximum Tweet identifier to a character vector
    cmax_id = char(max_id);
    % Search for Tweets
    s = search(c,tweetquery,'count',100,'max_id',cmax_id);
    % Retrieve Tweet text for each Tweet
    statuses = [statuses;s.Body.Data.statuses];
end
```

Retrieve the creation time and text of each Tweet. Retrieve the creation time for unstructured data by accessing it in a cell array of structures. For structured data, access the creation time by transposing the field in the structure array.

```
if iscell(statuses)
  % Unstructured data
    numTweets = length(statuses);             % Determine total number of Tweets
    tweetTimes = cell(numTweets,1);           % Allocate space for Tweet times and Tweet text
    tweetTexts = tweetTimes;
    for i = 1:numTweets
      tweetTimes{i} = statuses{i}.created_at; % Retrieve the time each Tweet was created
      tweetTexts{i} = statuses{i}.text;       % Retrieve the text of each Tweet
    end
else
    % Structured data
    tweetTimes = {statuses.created_at}';
    tweetTexts = {statuses.text}';
end
```

`tweetTimes` contains the creation time for each Tweet. `tweetTexts` contains the text for each Tweet.

Create the timetable `tweets` for all Tweets by using the text and creation time of each Tweet.

```
tweets = timetable(tweetTexts,'RowTimes', ...
    datetime(tweetTimes,'Format','eee MMM dd HH:mm:ss +SSSS yyyy'));
```

Conduct Sentiment Analysis on Tweets

Create a glossary of words that are associated with positive sentiment.

```
poskeywords = {'happy','great','good', ...
    'fast','optimized','nice','interesting','amazing','top','award', ...
    'winner','wins','cool','thanks','useful'};
```

`poskeywords` is a cell array of character vectors. Each character vector is a word that represents an instance of positive sentiment.

Search each Tweet for words in the positive sentiment glossary. Determine the total number of Tweets that contain a positive sentiment. Out of the total number of positive Tweets, determine the total number of Retweets.

```
% Determine the total number of Tweets
numTweets = height(tweets);
```

```
% Determine the positive Tweets
numPosTweets = 0;
numPosRTs = 0;
for i = 1:numTweets
    % Compare Tweet to positive sentiment glossary
    dJobs = contains(tweets.tweetTexts{i},poskeywords,'IgnoreCase',true);
    if dJobs
        % Increase total count of Tweets with positive sentiment by one
        numPosTweets = numPosTweets + 1;
        % Determine if positive Tweet is a Retweet
        RTs = strncmp('RT @',tweets.tweetTexts{i},4);
        if RTs
            % Increase total count of positive Retweets by one
            numPosRTs = numPosRTs + 1;
        end
    end
end
```

`numPosTweets` contains the total number of Tweets with positive sentiment.

`numPosRTs` contains the total number of Retweets with positive sentiment.

Create a glossary of words that are associated with negative sentiment.

```
negkeywords = {'sad','poor','bad','slow','weaken','mean','boring', ...
    'ordinary','bottom','loss','loser','loses','uncool', ...
    'criticism','useless'};
```

`negkeywords` is a cell array of character vectors. Each character vector is a word that represents an instance of negative sentiment.

Search each Tweet for words in the negative sentiment glossary. Determine the total number of Tweets that contain a negative sentiment. Out of the total number of negative Tweets, determine the total number of Retweets.

```
% Determine the negative Tweets
numNegTweets = 0;
numNegRTs = 0;
for i = 1:numTweets
    % Compare Tweet to negative sentiment glossary
    dJobs = contains(tweets.tweetTexts{i},negkeywords,'IgnoreCase',true);
    if dJobs
        % Increase total count of Tweets with negative sentiment by one
        numNegTweets = numNegTweets + 1;
        % Determine if negative Tweet is a Retweet
        RTs = strncmp('RT @',tweets.tweetTexts{i},4);
        if RTs
            numNegRTs = numNegRTs + 1;
        end
    end
end
```

`numNegTweets` contains the total number of Tweets with negative sentiment.

`numNegRTs` contains the total number of Retweets with negative sentiment.

Display Sentiment Analysis Results

Create a table with columns that contain:

- Number of Tweets
- Number of Tweets with positive sentiment
- Number of positive Retweets
- Number of Tweets with negative sentiment
- Number of negative Retweets

```
matlabTweetTable = table(numTweets,numPosTweets,numPosRTs,numNegTweets,numNegRTs, ...
    'VariableNames',{'Number_of_Tweets','Positive_Tweets','Positive_Retweets', ...
    'Negative_Tweets','Negative_Retweets'});
```

Display the table of Tweet data.

```
matlabTweetTable
```

```
matlabTweetTable =

  1×5 table

    Number_of_Tweets    Positive_Tweets    Positive_Retweets    Negative_Tweets    Negative_Retweets
    ________________    _______________    _________________    _______________    _________________

    11465               688                238                  201                96
```

Out of 11,465 total Tweets about the financial services industry in the last 7 days, 688 Tweets have positive sentiment and 201 Tweets have negative sentiment. Out of the positive Tweets, 238 Tweets are Retweets. Out of the negative Tweets, 96 are Retweets.

See Also

Functions

`search`

Objects

`twitter`

More About

- “Tweet Based on Retrieved Twitter Data” on page 9-6

External Websites

- Twitter REST API Endpoint Reference Documentation

Tweet Based on Retrieved Twitter Data

This example shows how to retrieve the number of followers for a Twitter account and Tweet about achieving a specific follower count. You can adapt this example to retrieve data from other Twitter REST API endpoints, such as collections, lists, and so on.

To run this example, you need Twitter credentials. To obtain these credentials, you must first log in to your Twitter account. Then, fill out the form in Create an application.

Connect to Twitter

Create a Twitter connection using your credentials. (The values in this example do not represent real Twitter credentials.)

```
consumerkey = 'abcdefghijklmnop123456789';
consumersecret = 'qrstuvwxyz123456789';
accesstoken = '123456789abcdefghijklmnop';
accesstokensecret = '123456789qrstuvwxyz';

c = twitter(consumerkey,consumersecret,accesstoken,accesstokensecret);
```

Check the Twitter connection. If the `StatusCode` property has the value `OK`, the connection is successful.

```
c.StatusCode
```

```
ans =

    OK
```

Retrieve Number of Followers

Set the Twitter base URL to access the `GET followers/ids` REST API endpoint. Search for a specific Twitter account using the Twitter connection object, base URL, and screen name. (The screen name in this example does not represent real Twitter data.)

```
baseurl = 'https://api.twitter.com/1.1/followers/ids.json';
sname = 'screenname';
d = getdata(c,baseurl,'screen_name',sname)
```

```
d =

  ResponseMessage with properties:

    StatusLine: 'HTTP/1.1 200 OK'
    StatusCode: OK
        Header: [1×25 matlab.net.http.HeaderField]
          Body: [1×1 matlab.net.http.MessageBody]
     Completed: 0
```

`d` is a `matlab.net.http.ResponseMessage` object. The `StatusCode` property shows `OK`, indicating a successful request.

Determine the number of followers for the specified account.

```
numfollowers = length(d.Body.Data.ids)
```

```
numfollowers =

    44
```

This account has 44 followers.

Post Tweet

Create the character vector `tweetString`, which specifies the Tweet to post. If the number of followers is greater than 25, then the Tweet indicates the screen name has more than 25 followers. Otherwise, it indicates that the screen name needs more followers.

```
if numfollowers > 25
    tweetString = [sname ' has more than 25 followers!'];
else
    tweetString = [sname ' needs more followers!'];
end
```

Set the Twitter base URL to access the `POST statuses/update` REST API endpoint.

```
baseurl = 'https://api.twitter.com/1.1/statuses/update.json';
```

Tweet about the number of followers using the Twitter connection object, base URL, and `tweetString`.

```
d = postdata(c,baseurl,'status',tweetString)
```

```
d =

  ResponseMessage with properties:

    StatusLine: 'HTTP/1.1 200 OK'
    StatusCode: OK
        Header: [1×22 matlab.net.http.HeaderField]
          Body: [1×1 matlab.net.http.MessageBody]
     Completed: 0
```

`d` is a `matlab.net.http.ResponseMessage` object. The `StatusCode` property shows `OK`, indicating a successful request.

See Also

Functions
`getdata` | `postdata`

Objects
`twitter`

More About

- “Conduct Sentiment Analysis Using Historical Tweets” on page 9-2

External Websites

- Twitter REST API Endpoint Reference Documentation

Tick History from Refinitiv Topics

Decide to Buy Shares with Intraday Data Using Tick History from Refinitiv

This example shows how to connect to Tick History from Refinitiv and trigger a buy decision for a single Reuters Instrument Code (RIC) using the trade price.

Create a Tick History from Refinitiv connection by using a user name and password. The appearance of the connection object `c` in the MATLAB workspace indicates a successful connection.

```
username = 'username';
password = 'password';
c = trth(username,password);
```

Retrieve intraday data for the IBM security. Using the `timeseries` function, retrieve the trade price from November 6, 2017, through November 7, 2017.

```
sec = ["IBM.N","Ric"];
fields = ["Trade - Price"];
startdate = datetime('11/06/2017','InputFormat','MM/dd/yyyy');
enddate = datetime('11/07/2017','InputFormat','MM/dd/yyyy');

d = timeseries(c,sec,fields,startdate,enddate);
```

Display the first three rows of intraday data.

```
head(d,3)
```

```
ans =

  3×5 timetable

           Time             x_RIC         Domain        GMTOffset     Type       Price
    ____________________    _____    ______________    _________    _______    ________

    06-Nov-2017 14:30:10    'IBM.N'    'Market Price'      '-5'       'Trade'    '151.68'
    06-Nov-2017 14:30:10    'IBM.N'    'Market Price'      '-5'       'Trade'    '151.66'
    06-Nov-2017 14:30:10    'IBM.N'    'Market Price'      '-5'       'Trade'    '151.73'
```

`d` is a timetable that contains these variables:

- Transaction date and time
- RIC
- Domain
- GMT time zone offset
- Transaction type
- Price

Assume a price threshold of $160. Determine if the trade price is less than $160. Set the buy indicator `buynow` to `true` when the threshold is met.

```
value = str2double(d.Price);
buynow = (value < 160);
```

Use the buy indicator to create a buy order of IBM shares in the trading system of your choice.

See Also

`trth` | `timeseries`

More About

- “Decide to Sell Shares with Historical Data Using Tick History from Refinitiv” on page 10-4

External Websites

- Refinitiv REST API Documentation

Decide to Sell Shares with Historical Data Using Tick History from Refinitiv

This example shows how to connect to Tick History from Refinitiv and trigger a sell decision for a single Reuters Instrument Code (RIC) using the closing price.

Create a Tick History from Refinitiv connection by using a user name and password. The appearance of the connection object `c` in the MATLAB workspace indicates a successful connection.

```
username = 'username';
password = 'password';
c = trth(username,password);
```

Retrieve historical data for the IBM security. Using the `history` function, retrieve the closing price from November 6, 2017, through November 7, 2017.

```
sec = ["IBM.N","Ric"];
fields = ["Last"];
startdate = datetime('11/06/2017','InputFormat','MM/dd/yyyy');
enddate = datetime('11/07/2017','InputFormat','MM/dd/yyyy');

d = history(c,sec,fields,startdate,enddate);
```

```
  2×1 timetable

      Time        Last
    __________    ______

    2017/11/06    150.84
    2017/11/07    151.35
```

Assume a price threshold of $150. Determine if the closing price is greater than $150. Set the sell indicator `sellnow` to `true` when the threshold is met.

```
sellnow = (d.Last > 150);
```

Use the sell indicator to create a sell order of IBM shares in the trading system of your choice.

See Also

`trth` | `history`

More About

- “Decide to Buy Shares with Intraday Data Using Tick History from Refinitiv” on page 10-2

External Websites

- Refinitiv REST API Documentation

Quandl Topics

Access Quandl Error Messages

When you request historical data from Quandl, sometimes the request returns an error instead of the historical data. Use this workflow to access Quandl error messages.

The `history` function returns errors in the `matlab.net.http.ResponseMessage` object. For example, suppose that you enter an invalid security name for the `s` input argument. The resulting output has this form:

```
d =

  ResponseMessage with properties:

    StatusLine: 'HTTP/1.1 404 Not Found'
    StatusCode: NotFound
        Header: [1×19 matlab.net.http.HeaderField]
          Body: [1×1 matlab.net.http.MessageBody]
     Completed: 0
```

Access the `Body` property using dot notation.

```
d.Body
```

```
ans =

  MessageBody with properties:

              Data: [1×1 struct]
           Payload: []
       ContentType: [1×1 matlab.net.http.MediaType]
    ContentCoding: [0×0 string]
```

To view the text of the error message, access the nested structure `quandl_error` stored in the `Data` property.

```
d.Body.Data.quandl_error
```

```
ans =

  struct with fields:

       code: 'QECx02'
    message: 'You have submitted an incorrect Quandl code. Please check your Quandl codes and try
```

Each error has a code and message associated with it. To view the code, access the `code` field. To view the error message text, access the `message` field. For example:

```
d.Body.Data.quandl_error.message
```

```
ans =

    'You have submitted an incorrect Quandl code. Please check your Quandl codes and try again.'
```

Refer to the error message to fix your code.

See Also

`quandl` | `history`

More About

- “Access Data in Nested Structures”

External Websites

- Quandl

Datastream Web Services Topics

Retrieve Datastream Web Services Historical Data

This example shows how to retrieve historical data from Datastream Web Services from Refinitiv. You must have Datastream Web Services credentials. For credentials, contact Datastream Web Services from Refinitiv.

Create Datastream Web Services Connection

Create a Datastream Web Services connection using your user name and password. `c` is the `datastreamws` connection object.

```
username = 'ABCDEF';
password = 'abcdef12345';
c = datastreamws(username,password);
```

Retrieve Monthly Historical Data

Adjust the display format to display currency.

```
format bank
```

Retrieve and display historical end-of-day price data from January 1, 2017, through December 31, 2017. Specify the `VOD` security and these fields:

- Opening price
- High price
- Last closing price

Specify a monthly period. `d` is a timetable with the date in the first variable and the fields in the subsequent variables.

```
sec = "VOD";
fields = ["PO";"PH";"P"];
startdate = datetime('01-01-2017','InputFormat','MM-dd-yyyy');
enddate = datetime('12-31-2017','InputFormat','MM-dd-yyyy');
period = 'M';
d = history(c,sec,fields,startdate,enddate,period)
```

```
d =

  12×3 timetable

          Time              PO        PH        P
    ____________________    ______    ______    ______

    01-Jan-2017 00:00:00       NaN       NaN    199.85
    01-Feb-2017 00:00:00    196.85    197.25    193.00
    01-Mar-2017 00:00:00    201.80    202.55    202.55
    01-Apr-2017 00:00:00    209.00    209.10    206.65
    01-May-2017 00:00:00       NaN       NaN    199.05
    01-Jun-2017 00:00:00    231.65    233.90    229.40
    01-Jul-2017 00:00:00    217.65    219.20    218.70
    01-Aug-2017 00:00:00    223.15    223.60    221.65
    01-Sep-2017 00:00:00    221.25    221.95    219.50
    01-Oct-2017 00:00:00    209.35    211.60    210.50
    01-Nov-2017 00:00:00    217.00    222.30    218.95
    01-Dec-2017 00:00:00    224.15    230.65    224.00
```

Retrieve Quarterly Historical Data

Retrieve and display historical end-of-day price data from January 1, 2017, through December 31, 2017. Specify the `VOD` security and these fields:

- Opening price
- High price
- Last closing price

Specify a quarterly period. `d` is a timetable with the date in the first variable and the fields in the subsequent variables.

```
sec = "VOD";
fields = ["PO";"PH";"P"];
startdate = datetime('01-01-2017','InputFormat','MM-dd-yyyy');
enddate = datetime('12-31-2017','InputFormat','MM-dd-yyyy');
period = 'Q';
d = history(c,sec,fields,startdate,enddate,period)
```

```
d =

  4×3 timetable

           Time              PO        PH        P
    ____________________    ______    ______    ______

    01-Jan-2017 00:00:00       NaN       NaN    199.85
    01-Apr-2017 00:00:00    209.00    209.10    206.65
    01-Jul-2017 00:00:00    217.65    219.20    218.70
    01-Oct-2017 00:00:00    209.35    211.60    210.50
```

Use the monthly and quarterly prices for each field to make investment decisions for the `VOD` security.

See Also

`datastreamws` | `history`

More About

- “Access Datastream Web Services Error Messages” on page 12-4

External Websites

- Datastream Web Services from Refinitiv REST Service

Access Datastream Web Services Error Messages

When you make a historical data request from Datastream Web Services from Refinitiv, sometimes the request returns an error instead of data. Use this workflow to access Datastream Web Services error messages.

The `history` function returns errors in the `matlab.net.http.ResponseMessage` object. For example, suppose that you enter an invalid security name for the `sec` input argument. The resulting output has this form:

```
d =

  ResponseMessage with properties:

    StatusLine: 'HTTP/1.1 200 OK'
    StatusCode: OK
        Header: [1×6 matlab.net.http.HeaderField]
          Body: [1×1 matlab.net.http.MessageBody]
     Completed: 0
```

Access the `Body` property using dot notation.

```
d.Body
```

```
ans =

  MessageBody with properties:

              Data: [1×1 struct]
           Payload: []
       ContentType: [1×1 matlab.net.http.MediaType]
    ContentCoding: [0×0 string]
```

To access the text of the error message, access the nested structure `DataResponse` stored in the `Data` property.

```
d.Body.Data.DataResponse
```

```
ans =

  struct with fields:

    AdditionalResponses: []
          DataTypeNames: []
         DataTypeValues: [3×1 struct]
                  Dates: []
            SymbolNames: []
                    Tag: ''
```

Then, access the `SymbolValues` field in the `DataTypeValues` structure array.

```
d.Body.Data.DataResponse.DataTypeValues(1).SymbolValues
```

```
ans =

  struct with fields:

    Currency: []
```

```
      Symbol: 'YYY'
        Type: 0
       Value: '$$ER: E100,INVALID CODE OR EXPRESSION ENTERED'
```

Fix your code based on the error message in the `Value` field.

See Also

`datastreamws` | `history`

More About

- “Retrieve Datastream Web Services Historical Data” on page 12-2
- “Access Data in Nested Structures”

External Websites

- Datastream Web Services from Refinitiv REST Service

IHS Markit Topics

Retrieve Factor Rank Data for Portfolio Selection

This example shows how to retrieve ranking data from IHS Markit for use in portfolio selection or an existing model. Retrieve percentile rank data for ticker security identifiers of a factor code. Then, use the rank information for portfolio selection or further analysis in an existing model. The example assumes that you have IHS Markit credentials. For credentials, see the IHS Markit website.

Create an IHS Markit connection using your user name and password. `c` is an `ihsmarkitrs` object.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password);
```

Retrieve signal information for the last 10 days using the IHS Markit connection. Specify the `ABR` factor code and `US Total Cap` universe. Also, specify the ticker security type and percentile data format. The percentile format provides factor ranking data. `d` is a table that contains signal information and the `date` and `data` variables.

```
code = 'ABR';
universeid = 'US Total Cap';
startdate = datetime('today')-10;
enddate = datetime('today');
identifier = 'ticker';
datatype = 'percentile';
d = signals(c,code,universeid,startdate,enddate,identifier,datatype);
```

Access the first few rows of ranking data for the first day in the date range by using the `data` variable.

```
data = d.data{1};
head(data)
```

```
ans =

  8×2 table

    ticker    value
    ______    _____

    'SVU'       1
    'LBY'       1
    'TLRY'      1
    'WIFI'      1
    'TCS'       1
    'AOBC'      1
    'TTD'       1
    'ZOES'      1
```

The variables of the resulting table are `ticker` and `value`. The `ticker` variable contains the ticker security identifiers. The `value` variable contains the factor ranking data.

Find all ticker security identifiers in `data` that have the most attractive value using rank values `1` through `10`. Create a table to store the rank values and perform an inner join to retrieve the most attractive securities. Display the last few attractive securities.

```
value = 1:10; % Define array of ranks 1 through 10
T = table(value','VariableNames',{'value'}); % Create table of the ranks in one variable
```

```
securities = innerjoin(data,T); % Perform inner join to find securities within the ranks
tail(securities)
```

```
ans =

  8×2 table

    ticker     value
    _______    _____

    'CDPYF'     10
    'CNXN'      10
    'DRNA'      10
    'PSX'       10
    'BRC'       10
    'ICHR'      10
    'MNLO'      10
    'MBI'       10
```

Use the factor rank data in your portfolio selection process or further analysis in your existing model.

See Also

`ihsmarkitrs` | `signals`

More About

- “IHS Markit Error Messages” on page 13-4

External Websites

- IHS Markit

IHS Markit Error Messages

This table describes how to address common errors you can encounter while working with IHS Markit.

Error Message	Probable Causes	Resolution
Unknown filter.	The specified factor code is invalid.	Specify a valid factor code.
Unknown Universe Name.	The specified universe name is invalid.	Specify a valid universe name.
Governor exceeded. More than 30 between *mm/DD/YYYY HH:MM:ss* and *mm/DD/YYYY HH:MM:ss*	For the specified date range, the returned data is too large.	Specify a shorter date range.

See Also

`ihsmarkitrs` | `factors` | `security` | `signals` | `universes`

More About

- “Retrieve Factor Rank Data for Portfolio Selection” on page 13-2

External Websites

- IHS Markit
- IHS Markit Research Signals REST Documentation

WDS Topics

Decide to Buy Shares Using Current and Historical WDS Data

This example shows how to connect to Wind Data Feed Services (WDS) and retrieve current and historical WDS data. The example then shows how to trigger a buy decision for a single security using the current high price. This example requires that you open and log in to the Wind Financial Terminal.

Connect to WDS

```
c = wind;
```

Retrieve Current Data for Security

Format output data for currency.

```
format bank
```

Using the `0001.HK` security, retrieve the current high and low prices.

```
s = '0001.HK';
f = ["high","low"];
d = getdata(c,s,f)
```

```
d=1×2 table
                HIGH      LOW
                _____    _____

    0001.HK    99.00    97.70
```

`d` is a table with one row for the single security. Each variable in the table corresponds to each specified field.

Retrieve Historical Data for Security

Using the same security, retrieve the high and low prices from August 1, 2017 through August 30, 2017.

```
f = ["high","low"];
startdate = datetime('2017-08-01');
enddate = datetime('2017-08-30');
h = history(c,s,f,startdate,enddate);
```

`h` is a timetable that contains one row for each trading day with the time and a variable for each specified field.

To create a threshold, you can analyze the historical data for the maximum and minimum high price.

```
max(h.HIGH)
```

```
ans = 108.9000
```

```
min(h.HIGH)
```

```
ans = 100.7000
```

Decide to Buy Shares

Assume a threshold of $100. Determine if the current high price is less than $100. Set the buy indicator `buynow` to `true` when the threshold is met.

```
buynow = (d.HIGH < 100);
```

Use the buy indicator and the `createorder` function to create a buy order of `0001.HK` shares.

Close WDS Connection

```
close(c)
```

See Also

`wind` | `getdata` | `history` | `close` | `createorder`

More About

- “Create Order Using Real-Time Snapshot WDS Data” on page 14-4

External Websites

- Wind Data Feed Services (WDS)

Create Order Using Real-Time Snapshot WDS Data

This example shows how to connect to Wind Data Feed Services (WDS), retrieve real-time snapshot data, and perform simple data analysis to make an investment decision. The example then shows how to log in to the WDS order management system, create an order, and query information about the order. This example requires that you open and log in to the Wind Financial Terminal.

Connect to WDS

```
c = wind;
```

Retrieve Snapshot Data

Format output data for currency.

```
format bank
```

Using the `600000.SH` security and the WDS connection, retrieve real-time snapshot data for the last price and volume fields.

```
s = '600000.SH';
f = {'rt_last','rt_vol'};

d = realtime(c,s,f)
```

```
d =

  1×3 timetable

            Time              Codes        RT_LAST       RT_VOL
    ____________________    ___________    _______    ____________

    05-Dec-2017 12:33:50    '600000.SH'     13.17     123796797.00
```

`d` is a timetable that contains a row for the security with the time and these variables:

- Security
- Last price
- Volume

Analyze Snapshot Price

Assume a price threshold of 12, specified in the CNY currency. Compare the snapshot price to the threshold. The sell indicator contains the logical value `1`.

```
sellnow = (d.RT_LAST > 12);
```

Set the direction of the order by using the sell indicator.

```
if (sellnow)
    direction = 'Sell';
else
    direction = 'Buy';
end
```

Create WDS Order

Log in to the WDS order management system using the WDS connection. Specify the broker, branch, user name, password, and account type.

```
broker = "0000";
branch = "0";
capitalaccount = "1234567891011";
password = "abcdefghi";
accttype = "SHSZ";
dlogin = tradelogin(c,broker,branch, ...
    capitalaccount,password,accttype);
```

Create a sell order of 100 shares of the `600000.SH` security using the WDS connection. Sell shares with the order price `13.17`, specified in the CNY currency. Use the `'LogonID'` name-value pair argument to specify the login identifier. Use the `'TradePassword'` name-value pair argument to specify the password.

```
price = '13.17';
quantity = '100';
logonid = '1';
password = "abcdefghi";
d = createorder(c,s,direction,price,quantity, ...
    'LogonID',logonid,'TradePassword',password)
```

```
d =

  1×8 table

    RequestID    SecurityCode    TradeSide    OrderPrice    OrderVolume    LogonID    ErrorCode
    _________    ____________    _________    __________    ___________    _______    _________

       20        '600000.sh'      'SELL'       '13.17'        '100'         '1'          0
```

`d` is a table with these variables:

- Request identifier
- Security code
- Direction
- Order price
- Order volume
- Login identifier
- Error code
- Error message

Query for the status of the executed order and display the status. The order status `'Normal'` indicates successful order execution.

```
d = query(c,'Order');
d.OrderStatus
```

```
d =

  'Normal'
```

Close WDS Connection

Log out from the WDS order management system using the login identifier returned by the `tradelogin` function.

```
logonid = dlogin.LogonID;
d = tradelogout(c,logonid);
```

Close the WDS connection.

```
close(c)
```

See Also

`wind` | `realtime` | `createorder` | `query` | `tradelogin` | `tradelogout` | `close`

More About

- “Decide to Buy Shares Using Current and Historical WDS Data” on page 14-2

External Websites

- Wind Data Feed Services (WDS)

Functions

blp

Bloomberg Desktop connection V3

Description

The `blp` function creates a `blp` object. The `blp` object represents a Bloomberg Desktop connection.

Other functions connect to different Bloomberg services: Bloomberg Server (`blpsrv`), and Bloomberg B-PIPE (`bpipe`). For details about these services, see “Comparing Bloomberg Connections” on page 2-4.

For details about Bloomberg connection requirements, see “Data Server Connection Requirements” on page 1-3. To ensure a successful Bloomberg connection, perform the required steps before executing `blp`. For details, see “Installing Bloomberg and Configuring Connections” on page 1-5.

Creation

Syntax

```
c = blp
c = blp(port,ip,timeout)
```

Description

`c = blp` creates a Bloomberg connection object that contains the Bloomberg Desktop connection. You need a Bloomberg Desktop software license for the machine running the Datafeed Toolbox and MATLAB.

`c = blp(port,ip,timeout)` sets the port and timeout properties, and uses the IP address of the local machine running Bloomberg to create a Bloomberg connection.

Caution To refer to a Bloomberg connection in other functions, use the connection object created by the `blp` function. Otherwise, using `blp` as an input argument opens multiple Bloomberg connections, causing unexpected behavior and exhausting memory resources.

Input Arguments

`ip` — IP address
`[ ]` (default) | character vector | string scalar

IP address that identifies the local machine running Bloomberg, specified as a character vector or string scalar.

Example: `'localhost'`

Data Types: `char` | `string`

Properties

`Session` — Bloomberg V3 session

Bloomberg V3 API Session object

This property is read-only.

Bloomberg V3 session, specified as a Bloomberg V3 API Session object.

Example: `[1x1 com.bloomberglp.blpapi.Session]`

`Port` — Port number of local machine

`[]` (default) | numeric scalar

Port number of the local machine running Bloomberg, specified as a numeric scalar.

Example: `8194`

Data Types: `double`

`IPAddress` — IP address of local machine

`[]` (default) | character vector

IP address of the local machine running Bloomberg, specified as a character vector.

The `blp` function sets this property using the `ip` input argument.

Example: `'localhost'`

Data Types: `char`

`TimeOut` — Timeout

numeric scalar

Timeout specifying the time in milliseconds that MATLAB attempts to connect to Bloomberg Desktop before timing out, specified as a numeric scalar.

Example: `10000`

Data Types: `double`

`DatetimeType` — Date and time data type

`''` (default) | `'datetime'`

Date and time data type, specified as one of these values.

Value	Description
`''` (default)	Return date and time values as MATLAB date numbers.
`'datetime'`	Return date and time values as a `datetime` array.

You can specify these values using a character vector or string (for example, `"datetime"`).

When you create a `blp` object, the `blp` function leaves this property unset. To retrieve data, you must set this property value manually at the command line or in a script using dot notation, for example:

```
c.DatetimeType = 'datetime';
```

Then, you can use these supported functions:

- `getbulkdata`
- `getdata`
- `history`
- `tahistory`
- `timeseries`

> **Note** If the `DataReturnFormat` property value is `'table'` and the `DatetimeType` property value is `'datetime'`, then the returned data is a table that contains date and time values as a `datetime` array. If the `DataReturnFormat` property value is an empty character vector, then setting the `DatetimeType` property to `'datetime'` returns date and time values for aggregated ticks and historical requests as MATLAB date numbers.

`DataReturnFormat` — Data return format
`'cell'` | `'structure'` | `'table'` | `'timetable'`

Data return format, specified as one of these values, which determine the data type of the returned data.

Value	Data Type of Returned Data
`'cell'`	`cell array`
`'table'`	`table`
`'timetable'`	`timetable`
`'structure'`	`structure`

Note The default data type of the returned data depends on the executed function. To specify the default data type, set the `DataReturnFormat` property to `''`. For default data types, see the supported function list.

You can specify these values using a character vector or string (for example, `"table"`).

When you create a `blp` object, the `blp` function leaves this property unset. To retrieve data, you must set this property value manually at the command line or in a script using dot notation, for example:

```
c.DataReturnFormat = 'structure';
```

Then, you can use these supported functions.

Supported Function	Valid Data Types for Returned Data
`category`	• `cell array` (default) • `structure` • `table`
`eqs`	• `cell array` (default) • `structure` • `table`
`fieldinfo`	• `cell array` (default) • `structure` • `table`
`fieldsearch`	• `cell array` (default) • `structure` • `table`
`lookup`	• `structure` (default) • `table`
`portfolio`	• `structure` (default) • `table`
`getbulkdata`	• `structure` (default) • `table` • `timetable`
`getdata`	• `structure` (default) • `table` • `timetable`
`history`	• numeric array (default) • `table` • `timetable`
`tahistory`	• `structure` (default) • `table` • `timetable`
`timeseries`	• `cell array` (default for raw tick data) • numeric array (default for interval tick data) • `table` • `timetable`

Note Regardless of the `DatetimeType` property value, if the `DataReturnFormat` property value is `'timetable'`, then the `getdata` and `getbulkdata` functions return a table that contains date and time values as `datetime` arrays.

Object Functions

Bloomberg Desktop Connection

close	Close Bloomberg connection V3
get	Properties of Bloomberg connection V3
isconnection	Determine Bloomberg connection V3

Bloomberg Desktop Data Retrieval

eqs	Equity screening data for Bloomberg connection V3
getbulkdata	Bulk data with header information for Bloomberg connection V3
getdata	Current data for Bloomberg connection V3
history	Historical data for Bloomberg connection V3
portfolio	Current portfolio data for Bloomberg connection V3
realtime	Real-time data for Bloomberg connection V3
stop	Unsubscribe real-time requests for Bloomberg connection V3
tahistory	Historical technical analysis for Bloomberg connection V3
timeseries	Intraday tick data for Bloomberg connection V3

Bloomberg Desktop Data Information

category	Field category search for Bloomberg connection V3
fieldinfo	Field information for Bloomberg connection V3
fieldsearch	Field search for Bloomberg connection V3
lookup	Find information about securities for Bloomberg connection V3

Examples

Connect to Bloomberg Desktop

First, create a Bloomberg® connection, and then retrieve current data for a security.

Create a connection to the Bloomberg Desktop.

```
c = blp
```

```
c =
  blp with properties:

            Session: [1×1 com.bloomberglp.blpapi.Session]
          IPAddress: 'localhost'
               Port: 8194
            TimeOut: 0
       DatetimeType: ''
    DataReturnFormat: ''
```

`c` is a Bloomberg connection object with these properties:

- Bloomberg V3 API Session object
- IP address of the local machine
- Port number of the local machine
- Number in milliseconds specifying how long MATLAB attempts to connect to Bloomberg Desktop before timing out
- Date and time data type
- Data return format

Request the last and open prices for Microsoft®.

```
format bank % Display data format for currency
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
[d,sec] = getdata(c,s,f)
```

```
d = struct with fields:
    LAST_PRICE: 72.28
          OPEN: 71.61


sec = 1×1 cell array
    {'MSFT US Equity'}
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the security in `sec`.

Close the Bloomberg Desktop connection.

```
close(c)
```

Connect to Bloomberg Desktop with Timeout

First, create a Bloomberg® connection with a timeout value, and then retrieve current data for a security.

Create a connection to the Bloomberg Desktop using the default port and IP address. Specify a timeout value of 10,000 milliseconds.

```
c = blp([],[],10000)
```

```
c =
  blp with properties:

            Session: [1×1 com.bloomberglp.blpapi.Session]
          IPAddress: 'localhost'
               Port: 8194
            TimeOut: 10000
       DatetimeType: ''
    DataReturnFormat: ''
```

The `blp` function creates a Bloomberg connection object `c` with these properties:

- Bloomberg V3 API Session object
- IP address of the local machine
- Port number of the local machine
- Number of milliseconds specifying how long MATLAB® attempts to connect to Bloomberg Desktop before timing out
- Date and time data type
- Data return format

Request the last and open prices for Microsoft®.

```
format bank % Display data format for currency
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
[d,sec] = getdata(c,s,f)
```

```
d = struct with fields:
    LAST_PRICE: 71.83
          OPEN: 71.61


sec = 1×1 cell array
    {'MSFT US Equity'}
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the security in `sec`.

Close the Bloomberg Desktop connection.

```
close(c)
```

Version History

Introduced in R2010a

See Also

Topics

“Connect to Bloomberg” on page 3-2
“Retrieve Current and Historical Data Using Bloomberg” on page 1-7
“Data Server Connection Requirements” on page 1-3
“Comparing Bloomberg Connections” on page 2-4
“Workflow for Bloomberg” on page 3-15

blpsrv

Bloomberg Server connection V3

Description

The `blpsrv` function creates a `blpsrv` object. The `blpsrv` object represents a Bloomberg Server connection.

Other functions connect to different Bloomberg services: Bloomberg Desktop (`blp`), and Bloomberg B-PIPE (`bpipe`). For details about these services, see “Comparing Bloomberg Connections” on page 2-4.

For details about Bloomberg connection requirements, see “Data Server Connection Requirements” on page 1-3. To ensure a successful Bloomberg connection, perform the required steps before executing `blpsrv`. For details, see “Installing Bloomberg and Configuring Connections” on page 1-5.

Creation

Syntax

```
c = blpsrv(uuid,ipaddress)
c = blpsrv(uuid,ipaddress,port)
c = blpsrv(uuid,ipaddress,port,timeout)
```

Description

`c = blpsrv(uuid,ipaddress)` creates a Bloomberg Server connection object `c` to the Bloomberg Server running on another machine, and sets the Uuid and IPAddress properties. You need a Bloomberg Server software license for the machine running the Bloomberg Server.

`c = blpsrv(uuid,ipaddress,port)` also sets the port property.

`c = blpsrv(uuid,ipaddress,port,timeout)` also sets the timeout property.

Caution To refer to a Bloomberg connection in other functions, use the connection object created by the `blpsrv` function. Otherwise, using `blpsrv` as an input argument opens multiple Bloomberg connections, causing unexpected behavior and exhausting memory resources.

Properties

`Uuid` — Bloomberg user identity UUID

numeric scalar

Bloomberg user identity UUID, specified as a numeric scalar. To find your UUID, enter `IAM` in the Bloomberg terminal and press **GO**.

Example: 12345678

Data Types: `double`

`User` — Bloomberg user
Bloomberg user identity object

This property is read-only.

Bloomberg user, specified as a Bloomberg user identity object.

Example: `[1x1 com.bloomberglp.blpapi.impl.aT]`

`Userip` — IP address of the machine running MATLAB
character vector

This property is read-only.

IP address of the machine running MATLAB, specified as a character vector.

Example: `'111.11.11.111'`

Data Types: `char`

`Session` — Bloomberg V3 session
Bloomberg V3 API Session object

This property is read-only.

Bloomberg V3 session, specified as a Bloomberg V3 API Session object.

Example: `[1x1 com.bloomberglp.blpapi.Session]`

`IPAddress` — Bloomberg Server IP address
character vector | string scalar

Bloomberg Server IP address, specified as a character vector or string scalar that identifies the machine running the Bloomberg Server.

Example: `'111.11.11.111'`

Data Types: `char` | `string`

`Port` — Port number
numeric scalar

Port number, specified as a numeric scalar that identifies the port number of the machine running the Bloomberg Server.

Example: `8194`

Data Types: `double`

`TimeOut` — Timeout
numeric scalar

Timeout specifying the time in milliseconds that MATLAB attempts to connect to the machine running the Bloomberg Server before timing out, specified as a numeric scalar.

Example: `10`

Data Types: `double`

`DatetimeType` — Date and time data type

`''` (default) | `'datetime'`

Date and time data type, specified as one of these values.

Value	Description
`''` (default)	Return date and time values as MATLAB date numbers.
`'datetime'`	Return date and time values as a `datetime` array.

You can specify these values using a character vector or string (for example, `"datetime"`).

When you create a `blpsrv` object, the `blpsrv` function leaves this property unset. To retrieve data, you must set this property value manually at the command line or in a script using dot notation, for example:

```
c.DatetimeType = 'datetime';
```

Then, you can use these supported functions:

- `getbulkdata`
- `getdata`
- `history`
- `tahistory`
- `timeseries`

> **Note** If the `DataReturnFormat` property value is `'table'` and the `DatetimeType` property value is `'datetime'`, then the returned data is a table that contains date and time values as a `datetime` array. If the `DataReturnFormat` property value is an empty character vector, then setting the `DatetimeType` property to `'datetime'` returns date and time values for aggregated ticks and historical requests as MATLAB date numbers.

`DataReturnFormat` — Data return format

`'cell'` | `'structure'` | `'table'` | `'timetable'`

Data return format, specified as one of these values, which determine the data type of the returned data.

Value	Data Type of Returned Data
`'cell'`	`cell array`
`'table'`	`table`
`'timetable'`	`timetable`

Value	Data Type of Returned Data
`'structure'`	`structure`

Note The default data type of the returned data depends on the executed function. To specify the default data type, set the `DataReturnFormat` property to `''`. For default data types, see the supported function list.

You can specify these values using a character vector or string (for example, `"table"`).

When you create a `blpsrv` object, the `blpsrv` function leaves this property unset. To retrieve data, you must set this property value manually at the command line or in a script using dot notation, for example:

```
c.DataReturnFormat = 'structure';
```

Then, you can use these supported functions.

Supported Function	Valid Data Types for Returned Data
`category`	• `cell array` (default) • `structure` • `table`
`eqs`	• `cell array` (default) • `structure` • `table`
`fieldinfo`	• `cell array` (default) • `structure` • `table`
`fieldsearch`	• `cell array` (default) • `structure` • `table`
`lookup`	• `structure` (default) • `table`
`portfolio`	• `structure` (default) • `table`
`getbulkdata`	• `structure` (default) • `table` • `timetable`

Supported Function	Valid Data Types for Returned Data
`getdata`	• `structure` (default) • `table` • `timetable`
`history`	• numeric array (default) • `table` • `timetable`
`tahistory`	• `structure` (default) • `table` • `timetable`
`timeseries`	• `cell array` (default for raw tick data) • numeric array (default for interval tick data) • `table` • `timetable`

Note Regardless of the `DatetimeType` property value, if the `DataReturnFormat` property value is `'timetable'`, then the `getdata` and `getbulkdata` functions return a table that contains date and time values as `datetime` arrays.

Object Functions

Bloomberg Server Connection

close Close Bloomberg connection V3
get Properties of Bloomberg connection V3
isconnection Determine Bloomberg connection V3

Bloomberg Server Data Retrieval

eqs Equity screening data for Bloomberg connection V3
getbulkdata Bulk data with header information for Bloomberg connection V3
getdata Current data for Bloomberg connection V3
history Historical data for Bloomberg connection V3
portfolio Current portfolio data for Bloomberg connection V3
realtime Real-time data for Bloomberg connection V3
stop Unsubscribe real-time requests for Bloomberg connection V3
tahistory Historical technical analysis for Bloomberg connection V3
timeseries Intraday tick data for Bloomberg connection V3

Bloomberg Server Data Information

category Field category search for Bloomberg connection V3
fieldinfo Field information for Bloomberg connection V3

fieldsearch	Field search for Bloomberg connection V3
lookup	Find information about securities for Bloomberg connection V3

Examples

Connect to Bloomberg Server

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = blpsrv(uuid,ipaddress)
```

```
c =

  blpsrv with properties:

               Uuid: 12345678
               User: [1x1 com.bloomberglp.blpapi.impl.aT]
             Userip: '111.11.11.112'
            Session: [1x1 com.bloomberglp.blpapi.Session]
          IPAddress: '111.11.11.111'
               Port: 8194
            TimeOut: 0
       DatetimeType: ''
    DataReturnFormat: ''
```

`blpsrv` connects to the machine running the Bloomberg Server using the default port number `8194`. `blpsrv` creates the Bloomberg Server connection object `c` with these properties:

- Bloomberg user identity UUID
- Bloomberg user identity object
- IP address of the machine running MATLAB
- Bloomberg V3 API Session object
- IP address of the machine running the Bloomberg Server
- Port number of the machine running the Bloomberg Server
- Number in milliseconds specifying how long MATLAB attempts to connect to the machine running the Bloomberg Server before timing out
- Date and time data type
- Data return format

Request the last and open prices for Microsoft.

```
format bank  % Display data format for currency
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
[d,sec] = getdata(c,s,f)
```

```
d =
    LAST_PRICE: 33.34
          OPEN: 33.60

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the name of the security in `sec`.

Close the Bloomberg Server connection.

```
close(c)
```

Connect to Bloomberg Server with Port Number

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.
- The port number of the machine running the Bloomberg Server is `8194`.

```
uuid = 12345678;
ipaddress = '111.11.11.111';
port = 8194;

c = blpsrv(uuid,ipaddress,port)

c =

  blpsrv with properties:

               Uuid: 12345678
               User: [1x1 com.bloomberglp.blpapi.impl.aT]
             Userip: '111.11.11.112'
            Session: [1x1 com.bloomberglp.blpapi.Session]
          IPAddress: '111.11.11.111'
               Port: 8194
            TimeOut: 0
        DatetimeType: ''
    DataReturnFormat: ''
```

`blpsrv` connects to the machine running the Bloomberg Server using the port number `8194` and creates the Bloomberg Server connection object `c` with these properties:

- Bloomberg user identity UUID
- Bloomberg user identity object
- IP address of the machine running MATLAB
- Bloomberg V3 API Session object
- IP address of the machine running the Bloomberg Server
- Port number of the machine running the Bloomberg Server

- Number in milliseconds specifying how long MATLAB attempts to connect to the machine running the Bloomberg Server before timing out
- Date and time data type
- Data return format

Request the last and open prices for Microsoft.

```
format bank  % Display data format for currency
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
[d,sec] = getdata(c,s,f)

d =
    LAST_PRICE: 33.34
          OPEN: 33.60

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the name of the security in `sec`.

Close the Bloomberg Server connection.

```
close(c)
```

Connect to Bloomberg Server with Timeout

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.
- The port number of the machine running the Bloomberg Server is your default port number.
- The timeout value is 10 milliseconds.

```
uuid = 12345678;
ipaddress = '111.11.11.111';
port = [];
timeout = 10;

c = blpsrv(uuid,ipaddress,port,timeout)

c =

  blpsrv with properties:

              Uuid: 12345678
              User: [1x1 com.bloomberglp.blpapi.impl.aT]
            Userip: '111.11.11.112'
           Session: [1x1 com.bloomberglp.blpapi.Session]
         IPAddress: '111.11.11.111'
              Port: 8194
           TimeOut: 10
```

```
        DatetimeType: ''
    DataReturnFormat: ''
```

`blpsrv` connects to the machine running the Bloomberg Server using the default port number `8194` and a timeout value of 10 milliseconds. `blpsrv` creates the Bloomberg Server connection object `c` with these properties:

- Bloomberg user identity UUID
- Bloomberg user identity object
- IP address of the machine running MATLAB
- Bloomberg V3 API Session object
- IP address of the machine running the Bloomberg Server
- Port number of the machine running the Bloomberg Server
- Number in milliseconds specifying how long MATLAB attempts to connect to the machine running the Bloomberg Server before timing out
- Date and time data type
- Data return format

Request the last and open prices for Microsoft.

```
format bank  % Display data format for currency
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
[d,sec] = getdata(c,s,f)

d =
    LAST_PRICE: 33.34
          OPEN: 33.60

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the name of the security in `sec`.

Close the Bloomberg Server connection.

```
close(c)
```

Version History

Introduced in R2014b

See Also

Topics

“Connect to Bloomberg” on page 3-2
“Data Server Connection Requirements” on page 1-3
“Comparing Bloomberg Connections” on page 2-4
“Workflow for Bloomberg” on page 3-15

bpipe

Bloomberg B-PIPE connection V3

Description

The `bpipe` function creates a `bpipe` object. The `bpipe` object represents a Bloomberg B-PIPE connection.

Other functions connect to different Bloomberg services: Bloomberg Desktop (`blp`), and Bloomberg Server (`blpsrv`). For details about these services, see “Comparing Bloomberg Connections” on page 2-4.

For details about Bloomberg connection requirements, see “Data Server Connection Requirements” on page 1-3. To ensure a successful Bloomberg connection, perform the required steps before executing `bpipe`. For details, see “Installing Bloomberg and Configuring Connections” on page 1-5.

Creation

Syntax

```
c = bpipe(authtype,appname,ipaddress,port)
c = bpipe(authtype,appname,ipaddress,port,timeout)
c = bpipe(authtype,appname,ipaddress,port,timeout,tlscred,tlspassword,
tlstrust)
```

Description

`c = bpipe(authtype,appname,ipaddress,port)` creates a Bloomberg B-PIPE connection object `c`, and sets these properties:

- authtype
- appname
- ipaddress
- port

`c = bpipe(authtype,appname,ipaddress,port,timeout)` also sets the timeout property.

`c = bpipe(authtype,appname,ipaddress,port,timeout,tlscred,tlspassword, tlstrust)` connects to a B-PIPE zero-footprint cloud solution using the specified credentials file, password, and trust file.

Caution To refer to a Bloomberg connection in other functions, use the connection object created by the `bpipe` function. Otherwise, using `bpipe` as an input argument opens multiple Bloomberg connections, causing unexpected behavior and exhausting memory resources.

Input Arguments

`tlscred` — Credentials file
character vector | string scalar

Credentials file, specified as a character vector or string scalar that contains the full path to the credentials file with the extension `pk12`. For details about the credentials file, contact Bloomberg.

Data Types: `char` | `string`

`tlspassword` — B-PIPE password
character vector | string scalar

B-PIPE password, specified as a character vector or string scalar. To obtain your B-PIPE password, contact Bloomberg.

Data Types: `char` | `string`

`tlstrust` — Trust file
character vector | string scalar

Trust file, specified as a character vector or string scalar that contains the full path to the trust file with the extension `pk7`. For details about the trust file, contact Bloomberg.

Data Types: `char` | `string`

Properties

`AppAuthType` — Application authentication type
`""` (default) | `"APPNAME_AND_KEY"`

This property is read-only.

Application authentication type, specified as one of these values:

- `""` — Bloomberg B-PIPE connection with Windows authentication
- `"APPNAME_AND_KEY"` — Bloomberg B-PIPE connection with application authentication

`AuthType` — Bloomberg user authentication type
`"OS_LOGON"` | `"APPLICATION_ONLY"`

Bloomberg user authentication type, specified as one of these values:

- `"OS_LOGON"` — Bloomberg B-PIPE connection with Windows authentication
- `"APPLICATION_ONLY"` — Bloomberg B-PIPE connection with application authentication

For details, see the *Bloomberg B-PIPE API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`AppName` — Application name
character vector | string

Application name, specified as a character vector or string that identifies the application you are using to connect to Bloomberg B-PIPE.

Example: `'appname'`

Data Types: `char` | `string`

`User` — Bloomberg user
Bloomberg user identity object

This property is read-only.

Bloomberg user, specified as a Bloomberg user identity object.

Example: `[1x1 com.bloomberglp.blpapi.impl.aT]`

`Session` — Bloomberg V3 session
Bloomberg V3 API Session object

This property is read-only.

Bloomberg V3 session, specified as a Bloomberg V3 API Session object.

Example: `[1x1 com.bloomberglp.blpapi.Session]`

`IPAddress` — IP address
character vector | cell array of character vectors | string | string array

IP address of the machine running the Bloomberg B-PIPE process, specified as a character vector, cell array of character vectors, string, or string array. A character vector or string identifies the machine running the Bloomberg B-PIPE process, whereas a cell array of character vectors or string array specifies multiple machines.

Example: `{'111.11.11.112'}`

Data Types: `char` | `cell` | `string`

`Port` — Port number
`[]` (default) | numeric scalar

Port number of the machine running the Bloomberg B-PIPE process, specified as a numeric scalar.

Example: `8194`

Data Types: `double`

`TimeOut` — Timeout
numeric scalar

Timeout specifying the time in milliseconds that MATLAB attempts to connect to the machine running the Bloomberg B-PIPE process before timing out, specified as a numeric scalar.

Example: `1000`

Data Types: `double`

`DatetimeType` — Date and time data type
`''` (default) | `'datetime'`

Date and time data type, specified as one of these values.

Value	Description
'' (default)	Return date and time values as MATLAB date numbers.
'datetime'	Return date and time values as a datetime array.

You can specify these values using a character vector or string (for example, "datetime").

When you create a bpipe object, the bpipe function leaves this property unset. To retrieve data, you must set this property value manually at the command line or in a script using dot notation, for example:

```
c.DatetimeType = 'datetime';
```

Then, you can use these supported functions:

- getbulkdata
- getdata
- history
- tahistory
- timeseries

Note If the DataReturnFormat property value is 'table' and the DatetimeType property value is 'datetime', then the returned data is a table that contains date and time values as a datetime array. If the DataReturnFormat property value is an empty character vector, then setting the DatetimeType property to 'datetime' returns date and time values for aggregated ticks and historical requests as MATLAB date numbers.

DataReturnFormat — Data return format
'cell' | 'structure' | 'table' | 'timetable'

Data return format, specified as one of these values, which determine the data type of the returned data.

Value	Data Type of Returned Data
'cell'	cell array
'table'	table
'timetable'	timetable
'structure'	structure

Note The default data type of the returned data depends on the executed function. To specify the default data type, set the `DataReturnFormat` property to `''`. For default data types, see the supported function list.

You can specify these values using a character vector or string (for example, `"table"`).

When you create a `bpipe` object, the `bpipe` function leaves this property unset. To retrieve data, you must set this property value manually at the command line or in a script using dot notation, for example:

```
c.DataReturnFormat = 'structure';
```

Then, you can use these supported functions.

Supported Function	Valid Data Types for Returned Data
`category`	• `cell array` (default) • `structure` • `table`
`eqs`	• `cell array` (default) • `structure` • `table`
`fieldinfo`	• `cell array` (default) • `structure` • `table`
`fieldsearch`	• `cell array` (default) • `structure` • `table`
`lookup`	• `structure` (default) • `table`
`portfolio`	• `structure` (default) • `table`
`getbulkdata`	• `structure` (default) • `table` • `timetable`
`getdata`	• `structure` (default) • `table` • `timetable`
`history`	• numeric array (default) • `table` • `timetable`

Supported Function	Valid Data Types for Returned Data
`tahistory`	• `structure` (default) • `table` • `timetable`
`timeseries`	• `cell array` (default for raw tick data) • numeric array (default for interval tick data) • `table` • `timetable`

Note Regardless of the `DatetimeType` property value, if the `DataReturnFormat` property value is `'timetable'`, then the `getdata` and `getbulkdata` functions return a table that contains date and time values as `datetime` arrays.

Object Functions

Bloomberg B-PIPE Connection

close	Close Bloomberg connection V3
get	Properties of Bloomberg connection V3
isconnection	Determine Bloomberg connection V3

Bloomberg B-PIPE Data Retrieval

eqs	Equity screening data for Bloomberg connection V3
getbulkdata	Bulk data with header information for Bloomberg connection V3
getdata	Current data for Bloomberg connection V3
history	Historical data for Bloomberg connection V3
portfolio	Current portfolio data for Bloomberg connection V3
realtime	Real-time data for Bloomberg connection V3
stop	Unsubscribe real-time requests for Bloomberg connection V3
tahistory	Historical technical analysis for Bloomberg connection V3
timeseries	Intraday tick data for Bloomberg connection V3

Bloomberg B-PIPE Data Information

category	Field category search for Bloomberg connection V3
fieldinfo	Field information for Bloomberg connection V3
fieldsearch	Field search for Bloomberg connection V3
lookup	Find information about securities for Bloomberg connection V3

Examples

Create Bloomberg B-PIPE Connection

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bpipe(authtype,appname,ipaddress,port)
```

```
c =

  bpipe with properties:

         AppAuthType: ''
            AuthType: 'OS_LOGON'
             AppName: []
                User: [1x1 com.bloomberglp.blpapi.impl.aT]
             Session: [1x1 com.bloomberglp.blpapi.Session]
           IPAddress: {'111.11.11.112'}
                Port: 8194.00
             TimeOut: 0
        DatetimeType: ''
    DataReturnFormat: ''
```

`bpipe` connects to the machine running Bloomberg B-PIPE at port number `8194`. `bpipe` creates the Bloomberg B-PIPE connection object `c` with these properties:

- Application authentication type
- Bloomberg user authentication type
- Application name
- Bloomberg user identity object
- Bloomberg V3 API Session object
- IP address of the machine running the Bloomberg B-PIPE process
- Port number of the machine running the Bloomberg B-PIPE process
- Number (in milliseconds) specifying how long MATLAB attempts to connect to the machine before timing out
- Date and time data type
- Data return format

Request the last and open prices for Microsoft.

```
format bank  % Display data format for currency
s = 'MSFT US Equity';
```

```
f = {'LAST_PRICE';'OPEN'};
[d,sec] = getdata(c,s,f)

d =
    LAST_PRICE: 33.34
          OPEN: 33.60

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the name of the security in `sec`.

Close the Bloomberg B-PIPE connection.

```
close(c)
```

Create Bloomberg B-PIPE Connection with Timeout

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.
- The timeout value is 1000 milliseconds.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;
timeout = 1000;

c = bpipe(authtype,appname,ipaddress,port,timeout)

c =

  bpipe with properties:

         AppAuthType: ''
            AuthType: 'OS_LOGON'
             AppName: []
                User: [1x1 com.bloomberglp.blpapi.impl.aT]
             Session: [1x1 com.bloomberglp.blpapi.Session]
           IPAddress: {'172.28.17.118'}
                Port: 8194.00
             TimeOut: 1000.00
        DatetimeType: ''
    DataReturnFormat: ''
```

`bpipe` connects to the machine running Bloomberg B-PIPE at port number `8194`. `bpipe` creates the Bloomberg B-PIPE connection object `c` with these properties:

- Application authentication type
- Bloomberg user authentication type
- Application name
- Bloomberg user identity object
- Bloomberg V3 API Session object
- IP address of the machine running the Bloomberg B-PIPE process
- Port number of the machine running the Bloomberg B-PIPE process
- Number (in milliseconds) specifying how long MATLAB attempts to connect to the machine before timing out
- Date and time data type
- Data return format

Request the last and open prices for Microsoft.

```
format bank  % Display data format for currency
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
[d,sec] = getdata(c,s,f)
```

```
d =
    LAST_PRICE: 33.34
          OPEN: 33.60

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the name of the security in `sec`.

Close the Bloomberg B-PIPE connection.

```
close(c)
```

Create Bloomberg B-PIPE Zero-Footprint Connection

Create a Bloomberg B-PIPE zero-footprint connection using the IP address of the machine running the Bloomberg B-PIPE process. This example assumes the following:

- The authentication is based on the application name when you set `authtype` to `'APPLICATION_ONLY'`.
- The application name is `'APP'`.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.
- The number (in milliseconds) specifying how long MATLAB attempts to connect to the machine before timing out is 1000.
- The full path of the credentials file is `C:\ABCDEFG.pk12`.
- The B-PIPE password is `12345`.
- The full path of the trust file is `C:\HIJKLM.pk7`.

```
authtype = 'APPLICATION_ONLY';
appname = 'APP';
ipaddress = {'111.11.11.112'};
port = 8194;
timeout = 1000;
tlscred = 'C:\ABCDEFG.pk12';
tlspassword = '12345';
tlstrust = 'C:\HIJKLM.pk7';

c = bpipe(authtype,appname,ipaddress,port, ...
    timeout,tlscred,tlspassword,tlstrust)

c =

  bpipe with properties:

         AppAuthType: 'APPNAME_AND_KEY'
            AuthType: 'APPLICATION_ONLY'
             AppName: 'APP'
                User: [1x1 com.bloomberglp.blpapi.impl.by]
             Session: [1x1 com.bloomberglp.blpapi.Session]
           IPAddress: {'111.11.11.112'}
                Port: 8194.00
             TimeOut: 1000.00
        DatetimeType: ''
    DataReturnFormat: ''
```

`bpipe` connects to the machine running Bloomberg B-PIPE at port number `8194`. The `bpipe` function creates the Bloomberg B-PIPE connection object `c` with these properties:

- Application authentication type
- Bloomberg user authentication type
- Application name
- Bloomberg user identity object
- Bloomberg V3 API Session object
- IP address of the machine running the Bloomberg B-PIPE process
- Port number of the machine running the Bloomberg B-PIPE process
- Number (in milliseconds) specifying how long MATLAB attempts to connect to the machine before timing out
- Date and time data type
- Data return format

Request the last and open prices for Microsoft.

```
format bank  % Display data format for currency
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
[d,sec] = getdata(c,s,f)

d =
    LAST_PRICE: 33.34
          OPEN: 33.60
```

```
sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the name of the security in `sec`.

Close the Bloomberg B-PIPE connection.

```
close(c)
```

Version History

Introduced in R2014b

See Also

Topics

“Connect to Bloomberg” on page 3-2
“Data Server Connection Requirements” on page 1-3
“Comparing Bloomberg Connections” on page 2-4
“Workflow for Bloomberg” on page 3-15

category

Field category search for Bloomberg connection V3

Syntax

```
d = category(c,f)
```

Description

`d = category(c,f)` returns category information given the search term `f`.

Examples

Search for Bloomberg Last Price Field

Create a Bloomberg® connection, and then request the category description of the last price field.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE® using `bpipe`.

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `category` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Request the Bloomberg category description of the last price field.

```
f = 'LAST_PRICE';
d = category(c,f);
```

Display the first three rows of the Bloomberg category description data in `d`.

```
d(1:3,:)
```

```
ans =

  3×5 table

    CATEGORY       ID                 MNEMONIC                          DESCRIPTION
    __________    _______    ____________________________    ____________________________

    'Analysis'    'OP179'    'THETA_LAST'                    'Theta Last Price'
    'Analysis'    'VM048'    'DDMX_PERCENT_CHANGE_LAST_PRICE'    'DDMX Percent Change Last Price
    'Analysis'    'YL005'    'YLD_CNV_LAST'                  'Last Yield To Convention'
```

The columns in `d` are:

- Category
- Field identifier
- Field mnemonic
- Field name
- Field data type

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

f — Search term

character vector | string scalar

Search term, specified as a character vector or string scalar to denote Bloomberg fields.

Data Types: `char` | `string`

Output Arguments

d — Category data

cell array (default) | structure | table

Category data, returned as an N-by-5 cell array, a structure, or a table.

The columns (or fields) of the data types are:

- Category
- Field identifier
- Field mnemonic
- Field name
- Field data type

The data type of the category data depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2010b

See Also

`blp` | `fieldinfo` | `fieldsearch` | `getdata` | `history` | `realtime` | `timeseries` | `close`

Topics

“Retrieve Bloomberg Current Data” on page 3-5
“Retrieve Current and Historical Data Using Bloomberg” on page 1-7
“Workflow for Bloomberg” on page 3-15

close

Close Bloomberg connection V3

Syntax

```
close(c)
```

Description

`close(c)` closes the Bloomberg connection V3 `c`.

Examples

Close the Bloomberg Connection

Create the Bloomberg connection object `c` using `blp`.

```
c = blp;
```

Alternatively, you can establish these connections:

- Bloomberg Server using `blpsrv`
- Bloomberg B-PIPE using `bpipe`

Close the Bloomberg connection using the Bloomberg connection object `c`.

```
close(c)
```

Input Arguments

c — Bloomberg connection

connection object

Bloomberg connection, specified as one of these connection objects:

- Bloomberg connection V3 created using `blp`
- Bloomberg Server connection created using `blpsrv`
- Bloomberg B-PIPE connection created using `bpipe`

Version History

Introduced in R2010a

See Also

`blp` | `blpsrv` | `bpipe`

Topics

“Connect to Bloomberg” on page 3-2
“Retrieve Bloomberg Current Data” on page 3-5
“Retrieve Bloomberg Historical Data” on page 3-7
“Retrieve Current and Historical Data Using Bloomberg” on page 1-7
“Retrieve Bloomberg Intraday Tick Data” on page 3-11
“Retrieve Bloomberg Real-Time Data” on page 3-13
“Workflow for Bloomberg” on page 3-15

eqs

Equity screening data for Bloomberg connection V3

Syntax

```
d = eqs(c,sname)
d = eqs(c,sname,stype)
d = eqs(c,sname,stype,languageid)
d = eqs(c,sname,stype,languageid,group)
d = eqs(c,sname,stype,languageid,group,'OverrideFields',ov)
```

Description

`d = eqs(c,sname)` returns equity screening data given the Bloomberg V3 session screen name `sname`.

`d = eqs(c,sname,stype)` also specifies the screen type `stype`.

`d = eqs(c,sname,stype,languageid)` also specifies the language identifier `languageid`.

`d = eqs(c,sname,stype,languageid,group)` also specifies the optional group identifier `group`.

`d = eqs(c,sname,stype,languageid,group,'OverrideFields',ov)` also specifies the Bloomberg override fields and values `ov`.

Examples

Retrieve Equity Screening Data for Screen

Create a Bloomberg® connection, and then retrieve frontier market stock data.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE® using `bpipe`.

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `eqs` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Retrieve equity screening data for the screen named `Frontier Market Stocks with 1 billion USD Market Caps`.

```
sname = 'Frontier Market Stocks with 1 billion USD Market Caps';
d = eqs(c,sname);
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  3×8 table

      Cntry                 Name                  IndGroup    MarketCap     Price_D_1     P_B
    __________    ___________________________    ________    __________    _________    _____

    'Venezuela'    'MERCANTIL SERVICIOS FINAN-A'    'Banks'    7.3424e+12      70088      278.25
    'Venezuela'    'BANCO DEL CARIBE-A'             'Banks'    2.0442e+12      24531      2321.8
    'Venezuela'    'BANCO PROVINCIAL'               'Banks'    1.2632e+12      11715      52.344
```

The columns in d are:

- Country name
- Company name
- Industry name
- Market capitalization
- Price
- Price-to-book ratio
- Price-earnings ratio
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Retrieve Equity Screening Data for a Screen Type

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve equity screening data for the screen called `Vehicle-Engine-Parts` and the screen type equal to `'GLOBAL'`.

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL');
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    'Ticker'        'Short Name'        'Market Cap'        'Price:D-1'    'P/E'
    'HON     US'    'HONEYWELL INTL'    [69451382784.00]    [    88.51]    [16.81]
    'CMI     US'    'CUMMINS INC'       [24799526912.00]    [   132.36]    [17.28]
```

```
  Columns 6 through 8

    'Total Return YTD'    'Revenue T12M'      'EPS T12M'
    [           42.43]    [38248998912.00]    [    4.11]
    [           24.43]    [17004999936.00]    [    7.57]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen. The first row contains column headers. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the connection.

```
close(c)
```

Retrieve Equity Screening Data for a Screen in German

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve equity screening data for the screen called `Vehicle-Engine-Parts`, the screen type equal to `'GLOBAL'`, and return data in German.

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL','GERMAN');
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
  Columns 1 through 5

    'Ticker'        'Kurzname'          'Marktkapitalisie...'    'Preis:D-1'    'KGV'
    'HON     US'    'HONEYWELL INTL'    [     69451382784.00]    [    88.51]    [16.81]
    'CMI     US'    'CUMMINS INC'       [     24799526912.00]    [   132.36]    [17.28]

  Columns 6 through 8

    'Gesamtertrag YTD'    'Erlös T12M'        'EPS T12M'
    [           42.43]    [38248998912.00]    [    4.11]
    [           24.43]    [17004999936.00]    [    7.57]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen. The first row contains column headers in German. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the connection.

```
close(c)
```

Retrieve Equity Screening Data for a Screen with a Specified Screen Folder Name

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve equity screening data for the Bloomberg screen called `Vehicle-Engine-Parts`, using the Bloomberg screen type `'GLOBAL'` and the language `'ENGLISH'`, and the Bloomberg screen folder name `'GENERAL'`.

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL','ENGLISH','GENERAL');
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    'Ticker'        'Short Name'        'Market Cap'        'Price:D-1'    'P/E'
    'HON     US'    'HONEYWELL INTL'    [69451382784.00]    [    88.51]    [16.81]
    'CMI     US'    'CUMMINS INC'       [24799526912.00]    [   132.36]    [17.28]

  Columns 6 through 8

    'Total Return YTD'    'Revenue T12M'      'EPS T12M'
    [           42.43]    [38248998912.00]    [    4.11]
    [           24.43]    [17004999936.00]    [    7.57]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen. The first row contains column headers. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio

- Total return year-to-date
- Revenue
- Earnings per share

Close the connection.

```
close(c)
```

Retrieve Equity Screening Data Using Override Fields

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve equity screening data as of a specified date using these input arguments. The override field `PiTDate` is equivalent to the flag `AsOf` in the Bloomberg Excel Add-In.

- Bloomberg connection `c`
- Bloomberg screen is `Vehicle-Engine-Parts`
- Bloomberg screen type is `'GLOBAL'`
- Language is `'ENGLISH'`
- Bloomberg screen folder name is `'GENERAL'`
- Override field `PiTDate` is September 9, 2014

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL','ENGLISH','GENERAL',...
    'OverrideFields',{'PiTDate','20140909'});
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    'Ticker'        'Short Name'        'Market Cap'    'Price:D-1'       'P/E'
    'HON US'     'HONEYWELL INTL'      [7.3919e+10]    [  94.4600]    [17.8087]
    'TSLA US'    'TESLA MOTORS'        [3.4707e+10]    [ 278.4800]    [    NaN]

  Columns 6 through 8

    'Total Return YTD'    'Revenue T12M'    'EPS T12M'
    [          4.8907]    [  3.9966e+10]    [  5.1600]
    [         85.1239]    [  2.4365e+09]    [ -1.3500]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen as of September 9, 2014. The first row contains column headers. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization

- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

`sname` — Screen name

character vector | string scalar

Screen name, specified as a character vector or string scalar to denote the Bloomberg V3 session screen name to execute. The screen can be a customized equity screen or one of the Bloomberg example screens accessed by using the **EQS <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

`stype` — Screen type

`'GLOBAL'` | `'PRIVATE'`

Screen type, specified as one of the two preceding values to denote the Bloomberg screen type. `'GLOBAL'` denotes a Bloomberg screen name and `'PRIVATE'` denotes a customized screen name. When using the optional `group` input argument, `stype` cannot be set to `'PRIVATE'` for customized screen names.

`languageid` — Language identifier

character vector | string scalar

Language identifier, specified as a character vector or string to denote the language for the returned data. This argument is optional.

Data Types: `char` | `string`

`group` — Group identifier

character vector | string scalar

Group identifier, specified as a character vector or string to denote the Bloomberg screen folder name accessed by using the **EQS <GO>** option from the Bloomberg terminal. This argument is optional. When using this argument, `stype` cannot be set to `'PRIVATE'` for customized screen names.

Data Types: `char` | `string`

`ov` — Bloomberg override field values

cell array

Bloomberg override field values, specified as an n-by-2 cell array. The first column of the cell array is the override field. The second column is the override value.

Example: `{'PiTDate','20140909'}`

Data Types: `cell`

Output Arguments

d — Equity screening data

cell array (default) | structure | table

Equity screening data, returned as a cell array, structure, or table. The data type of the equity screening data depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2012b

See Also

`blp` | `getdata` | `tahistory` | `close`

Topics

“Retrieve Bloomberg Current Data” on page 3-5
“Retrieve Current and Historical Data Using Bloomberg” on page 1-7
“Workflow for Bloomberg” on page 3-15

fieldinfo

Field information for Bloomberg connection V3

Syntax

```
d = fieldinfo(c,f)
```

Description

`d = fieldinfo(c,f)` returns field information on the Bloomberg V3 connection object `c` given the field mnemonic `f`.

Examples

Retrieve Information for Last Price Field

Create a Bloomberg® connection, and then retrieve information for the last price field.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE® using `bpipe`.

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `fieldinfo` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Retrieve the Bloomberg field information for the `LAST_PRICE` field.

```
f = 'LAST_PRICE';
d = fieldinfo(c,f);
```

Display the last four columns in the returned Bloomberg information.

```
d(:,2:5)
```

```
ans =

  1×4 table

      ID          MNEMONIC             DESCRIPTION           DATATYPE
    _______    _____________    ________________________    ________

    'RQ005'    'LAST_PRICE'    'Last Trade/Last Price'    'Double'
```

The columns in `d` are:

- Field identifier
- Field mnemonic
- Field name
- Field data type

You can also access the Bloomberg help information in the first column.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

f — Field mnemonic

character vector | string scalar

Field mnemonic, specified as a character vector or string scalar that denotes the Bloomberg field information to retrieve.

Data Types: `char` | `string`

Output Arguments

d — Field information

cell array (default) | structure | table

Field information, returned as an N-by-5 cell array, a structure, or a table.

The columns (or fields) of the data types are:

- Field help
- Field identifier
- Field mnemonic
- Field name
- Field data type

The data type of the field information depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2010b

See Also

`blp` | `category` | `fieldsearch` | `getdata` | `history` | `realtime` | `timeseries` | `close`

Topics

“Retrieve Bloomberg Current Data” on page 3-5
“Retrieve Current and Historical Data Using Bloomberg” on page 1-7
“Workflow for Bloomberg” on page 3-15

fieldsearch

Field search for Bloomberg connection V3

Syntax

```
d = fieldsearch(c,f)
```

Description

`d = fieldsearch(c,f)` returns field information on the Bloomberg V3 connection object `c` given the search term `f`.

Examples

Search for Last Price Field Information

Create a Bloomberg® connection, and then return information for the last price field.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE® using `bpipe`.

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `fieldsearch` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Return information for the search term `LAST_PRICE`.

```
f = 'LAST_PRICE';
d = fieldsearch(c,f);
```

Display the first three rows of the field information in `d`.

```
d(1:3,:)
```

```
ans =

  3×5 table

          CATEGORY             ID            MNEMONIC                DESCRIPTION
    ___________________________    _______    ____________________    ___________________________________

    'Market Activity/Last'       'PR005'    'PX_LAST'               'Last Price'
    'Market Activity/Last'       'RQ005'    'LAST_PRICE'            'Last Trade/Last Price'
    'Market Activity/Last'       'PR910'    'CRNCY_ADJ_PX_LAST'     'Currency Adjusted Last Price'
```

The columns in `d` are:

- Category
- Field identifier
- Field mnemonic
- Field name
- Field data type

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

`f` — Search term

character vector | string scalar

Search term, specified as a character vector or string scalar that denotes the Bloomberg field descriptive data to retrieve.

Data Types: `char` | `string`

Output Arguments

`d` — Field data

cell array (default) | structure | table

Field data, returned as an N-by-5 cell array, a structure, or a table.

The columns (or fields) of the data types are:

- Category
- Field identifier
- Field mnemonic
- Field name
- Field data type

The data type of the field data depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2010b

See Also

`blp` | `category` | `fieldinfo` | `getdata` | `history` | `realtime` | `timeseries` | `close`

Topics

“Retrieve Bloomberg Current Data” on page 3-5
“Retrieve Current and Historical Data Using Bloomberg” on page 1-7
“Workflow for Bloomberg” on page 3-15

get

Properties of Bloomberg connection V3

Syntax

```
v = get(c)
v = get(c,properties)
```

Description

`v = get(c)` returns a structure where each field name is the name of a property of `c` and each field contains the value of that property.

`v = get(c,properties)` returns the value of the specified properties `properties` for the Bloomberg V3 connection object.

Examples

Retrieve Bloomberg Connection Properties

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the Bloomberg connection properties.

```
v = get(c)
```

```
v =

      session: [1x1 com.bloomberglp.blpapi.Session]
    ipaddress: 'localhost'
         port: 8194
      timeout: 0
```

`v` is a structure containing the Bloomberg session object, IP address, port number, and timeout value.

Close the Bloomberg connection.

```
close(c)
```

Retrieve One Bloomberg Connection Property

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the port number from the Bloomberg connection object by specifying `'port'` as a character vector.

```
property = 'port';
v = get(c,property)
```

```
v =

        8194
```

`v` is a double that contains the port number of the Bloomberg connection object.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Two Bloomberg Connection Properties

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Create a cell array `properties` with character vectors `'session'` and `'port'`. Retrieve the Bloomberg session object and port number from the Bloomberg connection object.

```
properties = {'session','port'};
v = get(c,properties)
```

```
v =

    session: [1x1 com.bloomberglp.blpapi.Session]
       port: 8194
```

`v` is a structure containing the Bloomberg session object and port number.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

`properties` — Property names

character vector | string scalar | cell array of character vectors | string array

Property names, specified as a character vector, string scalar, cell array of character vectors, or string array containing Bloomberg connection property names. The property names are `session`, `ipaddress`, `port`, and `timeout`.

Data Types: `char` | `cell` | `string`

Output Arguments

v — Bloomberg connection properties

numeric scalar | character vector | object | structure

Bloomberg connection properties, returned as these data types depending on the requested properties.

Requested Properties	Data Type
Port number or timeout	Numeric scalar
IP address	Character vector
Bloomberg session	Object
All properties	Structure

Version History

Introduced in R2010a

See Also

`getdata` | `history` | `realtime` | `timeseries` | `blp` | `close`

Topics

“Connect to Bloomberg” on page 3-2
“Workflow for Bloomberg” on page 3-15

getbulkdata

Bulk data with header information for Bloomberg connection V3

Syntax

```
d = getbulkdata(c,s,f)
d = getbulkdata(c,s,f,o,ov)
d = getbulkdata(c,s,f,o,ov,Name,Value)
[d,sec] = getbulkdata( ___ )
```

Description

`d = getbulkdata(c,s,f)` returns the bulk data for the fields `f` for the security list `s`.

`d = getbulkdata(c,s,f,o,ov)` returns the bulk data using the override fields `o` with corresponding override values `ov`.

`d = getbulkdata(c,s,f,o,ov,Name,Value)` returns the bulk data with additional options specified by one or more name-value pair arguments for Bloomberg request settings.

`[d,sec] = getbulkdata( ___ )` additionally returns the security list `sec` using any of the input argument combinations in the previous syntaxes.

Examples

Return Specific Field for Given Security

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Return the dividend history for IBM.

```
security = 'IBM US Equity';
field = 'DVD_HIST'; % Dividend history field

[d,sec] = getbulkdata(c,security,field)
```

```
d =

    DVD_HIST: {{149x7 cell}}

sec =

    'IBM US Equity'
```

`d` is a structure with one field that contains a cell array with the returned bulk data. `sec` contains the IBM security name.

Display the dividend history with the associated header information by accessing the structure field `DVD_HIST`. This field is a cell array that contains one cell array. The nested cell array contains the dividend history data. Access the contents of the nested cell using cell array indexing.

```
d.DVD_HIST{1}
```

```
ans =

  Columns 1 through 6

    'Declared Date'    'Ex-Date'    'Record Date'    'Payable Date'    'Dividend Amount'    'Dividend Frequency'
    [       735536]    [ 735544]    [     735546]    [      735578]    [         0.95]      'Quarter'
    [       735445]    [ 735453]    [     735455]    [      735487]    [         0.95]      'Quarter'
    [       735354]    [ 735362]    [     735364]    [      735395]    [         0.95]      'Quarter'
    ...

  Column 7

    'Dividend Type'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
    ...
```

The first row of the dividend history data is the header information that describes the contents of each column.

Close the connection.

```
close(c)
```

Return Specific Field Using Override Values

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Return the dividend history for IBM with dividend dates from January 1, 2004, through January 1, 2005.

```
security = 'IBM US Equity';
field = 'DVD_HIST';                          % Dividend history field
override = {'DVD_START_DT','DVD_END_DT'};  % Dividend start and
                                             % End dates
overridevalues = {'20040101','20050101'};

[d,sec] = getbulkdata(c,security,field,override,overridevalues)
```

```
d =

    DVD_HIST: {{5x7 cell}}

sec =

    'IBM US Equity'
```

`d` is a structure with one field that contains a cell array with the returned bulk data. `sec` contains the IBM security name.

Display the dividend history with the associated header information by accessing the structure field `DVD_HIST`. This field is a cell array that contains one cell array. The nested cell array contains the dividend history data. Access the contents of the nested cell using cell array indexing.

```
d.DVD_HIST{1}
```

```
ans =

  Columns 1 through 6

    'Declared Date'    'Ex-Date'    'Record Date'    'Payable Date'    'Dividend Amount'    'Dividend Frequency'
    [       732246]    [ 732259]    [     732261]    [      732291]    [         0.18]      'Quarter'
    [       732155]    [ 732165]    [     732169]    [      732200]    [         0.18]      'Quarter'
    [       732064]    [ 732073]    [     732077]    [      732108]    [         0.18]      'Quarter'
    [       731973]    [ 731983]    [     731987]    [      732016]    [         0.16]      'Quarter'

  Column 7

    'Dividend Type'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
```

The first row of the dividend history data is the header information that describes the contents of each column.

Close the connection.

```
close(c)
```

Return Specific Field Using Name-Value Pair Arguments

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Return the closing price and dividend history for IBM with dividend dates from January 1, 2004 through January 1, 2005. Specify the data return format as a character vector by setting the name-value pair argument `'returnFormattedValue'` to `'true'`.

```
security = 'IBM US Equity';
fields = {'LAST_PRICE','DVD_HIST'};        % Closing price and
                                           % Dividend history fields
override = {'DVD_START_DT','DVD_END_DT'};  % Dividend start and
                                           % End dates
overridevalues = {'20040101','20050101'};

[d,sec] = getbulkdata(c,security,fields,override,overridevalues,...
                      'returnFormattedValue',true)
```

```
d =

      DVD_HIST: {{5x7 cell}}
    LAST_PRICE: {'188.74'}

sec =
```

```
    'IBM US Equity'
```

`d` is a structure with two fields. The first field `DVD_HIST` contains a cell array with the dividend historical data as a cell array. The second field `LAST_PRICE` contains a cell array with the closing price as a character vector. `sec` contains the IBM security name.

Display the closing price.

```
d.LAST_PRICE
```

```
ans =

    '188.74'
```

Display the dividend history with the associated header information by accessing the structure field `DVD_HIST`. This field is a cell array that contains one cell array. The nested cell array contains the dividend history data. Access the contents of the nested cell using cell array indexing.

```
d.DVD_HIST{1}
```

```
ans =

  Columns 1 through 6

    'Declared Date'    'Ex-Date'    'Record Date'    'Payable Date'    'Dividend Amount'    'Dividend Frequency'
    [       732246]    [ 732259]    [     732261]    [      732291]    [         0.18]      'Quarter'
    [       732155]    [ 732165]    [     732169]    [      732200]    [         0.18]      'Quarter'
    [       732064]    [ 732073]    [     732077]    [      732108]    [         0.18]      'Quarter'
    [       731973]    [ 731983]    [     731987]    [      732016]    [         0.16]      'Quarter'

  Column 7

    'Dividend Type'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
```

The first row of the dividend history data is the header information that describes the contents of each column.

Close the connection.

```
close(c)
```

Return Bulk Data as Table with Datetime

Create a Bloomberg® connection, and then request dividend history data. The `getbulkdata` function returns data for dates as a `datetime` array.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE® using `bpipe`.

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `getbulkdata` function returns data as a structure.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Return the dividend history for IBM®.

```
s = 'IBM US Equity';
f = 'DVD_HIST'; % Dividend history field

d = getbulkdata(c,s,f);
```

Display the first three rows of the table.

```
d.DVD_HIST{1}(1:3,:)
```

```
ans =

  3×7 table

       DeclaredDate            ExmDate             RecordDate            PayableDate
    ____________________    ____________________    ____________________    ____________

    31-Oct-2017 00:00:00    09-Nov-2017 00:00:00    10-Nov-2017 00:00:00    09-Dec-2017 00:00:00
    25-Jul-2017 00:00:00    08-Aug-2017 00:00:00    10-Aug-2017 00:00:00    09-Sep-2017 00:00:00
    25-Apr-2017 00:00:00    08-May-2017 00:00:00    10-May-2017 00:00:00    10-Jun-2017 00:00:00
```

Display three declared dates. The `DeclaredDate` variable is a `datetime` array.

```
d.DVD_HIST{1}.DeclaredDate(1:3)
```

```
ans =

  3×1 datetime array

   31-Oct-2017 00:00:00
   25-Jul-2017 00:00:00
   25-Apr-2017 00:00:00
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

`f` — Bloomberg data fields
character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

`o` — Bloomberg override field
`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field name. A cell array of character vectors or string array denotes multiple Bloomberg override field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `'END_DT'`

Data Types: `char` | `cell` | `string`

`ov` — Bloomberg override field value
`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field value, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field value. A cell array of character vectors or string array denotes multiple Bloomberg override field values. Use this field value to filter the Bloomberg data result set.

Example: `'20100101'`

Data Types: `char` | `cell` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'returnFormattedValue',true`

`returnEids` — Entitlement identifiers
true | false

Entitlement identifiers, specified as the comma-separated pair consisting of `'returnEids'` and a Boolean. `true` adds a name and value for the entitlement identifier (EID) date to the return data.

Data Types: `logical`

`returnFormattedValue` — Return format
true | false

Return format, specified as the comma-separated pair consisting of `'returnFormattedValue'` and a Boolean. `true` forces all data to be returned as the data type character vector.

Data Types: `logical`

`useUTCTime` — Date time format
true | false

Date time format, specified as the comma-separated pair consisting of `'useUTCTime'` and a Boolean. `true` returns date and time values as Coordinated Universal Time (UTC) and `false` defaults to the Bloomberg **TZDF <GO>** settings of the requestor.

Data Types: `logical`

`forcedDelay` — Latest reference data
true | false

Latest reference data, specified as the comma-separated pair consisting of `'forcedDelay'` and a Boolean. `true` returns the latest data up to the delay period specified by the exchange for the security.

Data Types: `logical`

Output Arguments

`d` — Bloomberg data
structure (default) | table | timetable

Bloomberg data, returned as a structure, table, or timetable. The data type of the Bloomberg data depends on the DataReturnFormat and DatetimeType properties of the connection object. For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`sec` — Security list
cell array of character vectors

Security list, returned as a cell array of character vectors for the corresponding securities in `s`. The contents of `sec` are identical in value and order to `s`. You can return securities with any of the following identifiers:

- `buid`
- `cats`
- `cins`
- `common`
- `cusip`
- `isin`
- `sedol1`

- `sedol2`
- `sicovam`
- `svm`
- `ticker` (default)
- `wpk`

Version History

Introduced in R2014b

See Also

`blp` | `getdata` | `history` | `realtime` | `timeseries` | `close`

Topics

“Retrieve Bloomberg Current Data” on page 3-5
“Retrieve Current and Historical Data Using Bloomberg” on page 1-7
“Workflow for Bloomberg” on page 3-15

getdata

Current data for Bloomberg connection V3

Syntax

```
d = getdata(c,s,f)
d = getdata(c,s,f,o,ov)
d = getdata(c,s,f,o,ov,Name,Value)
[d,sec] = getdata( ___ )
```

Description

`d = getdata(c,s,f)` returns the data for the fields `f` for the security list `s`. `getdata` accesses the Bloomberg reference data service.

`d = getdata(c,s,f,o,ov)` returns the data using the override fields `o` with corresponding override values `ov`.

`d = getdata(c,s,f,o,ov,Name,Value)` returns the data using name-value pair arguments for additional Bloomberg request settings.

`[d,sec] = getdata( ___ )` additionally returns the security list `sec` using any of the input argument combinations in the previous syntaxes.

Examples

Last and Open Price for Security

First, create a Bloomberg Desktop connection. Then, request last and open prices for a security. The current data you see when running this code can differ from the output data here.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Request last and open prices for Microsoft.

```
[d,sec] = getdata(c,'MSFT US Equity',{'LAST_PRICE';'OPEN'})
```

```
d =
    LAST_PRICE: 33.3401
          OPEN: 33.6000

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the security in `sec`.

Close the connection.

```
close(c)
```

Specified Fields Given Override Fields and Values

First, create a Bloomberg Desktop connection. Then, request data for specific fields for a security using an override field and value. The current data you see when running this code can differ from the output data here.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Request data for Bloomberg fields `'YLD_YTM_ASK'`, `'ASK'`, and `'OAS_SPREAD_ASK'` when the Bloomberg field `'OAS_VOL_ASK'` is `'14.000000'`.

```
[d,sec] = getdata(c,'030096AF8 Corp',...
   {'YLD_YTM_ASK','ASK','OAS_SPREAD_ASK','OAS_VOL_ASK'},...
   {'OAS_VOL_ASK'},{'14.000000'})
```

```
d =
       YLD_YTM_ASK: 5.6763
               ASK: 120.7500
    OAS_SPREAD_ASK: 307.9824
       OAS_VOL_ASK: 14

sec =
    '030096AF8 Corp'
```

`getdata` returns a structure `d` with the resulting values for the requested fields.

Close the connection.

```
close(c)
```

Request for Security Using CUSIP Number

First, create a Bloomberg Desktop connection. Then, use the CUSIP number for a security to request last price. The current data you see when running this code can differ from the output data here.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Request the last price for IBM with the CUSIP number.

```
d = getdata(c,'/cusip/459200101','LAST_PRICE')
```

```
d =
    LAST_PRICE: 182.5100
```

`getdata` returns a structure `d` with the last price.

Close the connection.

```
close(c)
```

Last Price for Security with Pricing Source

First, create a Bloomberg Desktop connection. Then, request the last price for a security. Specify the security using the CUSIP number with a pricing source. The current data you see when running this code can differ from the output data here.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Specify IBM with the CUSIP number and the pricing source `BGN` after the `@` symbol.

```
d = getdata(c,'/cusip/459200101@BGN','LAST_PRICE')
```

```
d =
    LAST_PRICE: 186.81
```

`getdata` returns a structure `d` with the last price.

Close the connection.

```
close(c)
```

Constituent Weights Using Date Override

First, create a Bloomberg Desktop connection. Then, request the constituent weights of an index using a date override. The current data you see when running this code can differ from the output data here.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Return the constituent weights for the Dow Jones Index as of January 1, 2010, using a date override with the required date format `YYYYMMDD`.

```
d = getdata(c,'DJX Index','INDX_MWEIGHT','END_DT','20100101')
```

```
d =
    INDX_MWEIGHT: {{30x2 cell}}
```

`getdata` returns a structure `d` with a cell array where the first column is the index and the second column is the constituent weight.

Display the constituent weights for each index.

```
d.INDX_MWEIGHT{1,1}
```

```
ans =
    'AA UN'      [1.1683]
    'AXP UN'     [2.9366]
    'BA UN'      [3.9229]
    'BAC UN'     [1.0914]
    ...
```

Close the connection.

```
close(c)
```

Current Data and Dates as Table with Datetime

Create a Bloomberg® connection, and then request current data for specific fields. The `getdata` function returns data for dates as a `datetime` array.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE® using `bpipe`.

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `getdata` function returns data as a structure.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Request current data for these fields:

- Last update date
- Last price
- Number of trades
- Previous real-time trading date

```
s = 'IBM US Equity';
f = {'LAST_UPDATE_DT','LAST_PRICE', ...
    'NUM_TRADES_RT','PREV_TRADING_DT_REALTIME'};
d = getdata(c,s,f)
```

```
d =

  1×4 table
```

LAST_UPDATE_DT	LAST_PRICE	NUM_TRADES_RT	PREV_TRADING_DT_REALTIME
21-Dec-2017 00:00:00	152.2	24846	20-Dec-2017 00:00:00

Display the last update date. This date is a `datetime` array.

```
d.LAST_UPDATE_DT
```

```
ans =

  datetime

   21-Dec-2017 00:00:00
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

f — Bloomberg data fields

character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

o — Bloomberg override field

`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field name. A cell array

of character vectors or string array denotes multiple Bloomberg override field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `'END_DT'`

Data Types: `char` | `cell` | `string`

`ov` — Bloomberg override field value
`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field value, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field value. A cell array of character vectors or string array denotes multiple Bloomberg override field values. Use this field value to filter the Bloomberg data result set.

Example: `'20100101'`

Data Types: `char` | `cell` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'returnEids',true`

`returnEids` — Entitlement identifiers
true | false

Entitlement identifiers, specified as the comma-separated pair consisting of `'returnEids'` and a Boolean. `true` adds a name and value for the entitlement identifier (EID) date to the return data.

Data Types: `logical`

`returnFormattedValue` — Return format
true | false

Return format, specified as the comma-separated pair consisting of `'returnFormattedValue'` and a Boolean. `true` forces all data to be returned as the data type character vector.

Data Types: `logical`

`useUTCTime` — Date time format
true | false

Date time format, specified as the comma-separated pair consisting of `'useUTCTime'` and a Boolean. `true` returns date and time values as Coordinated Universal Time (UTC) and `false` defaults to the Bloomberg **TZDF <GO>** settings of the requestor.

Data Types: `logical`

`forcedDelay` — Latest reference data
true | false

Latest reference data, specified as the comma-separated pair consisting of `'forcedDelay'` and a Boolean. `true` returns the latest data up to the delay period specified by the exchange for the security.

Data Types: `logical`

Output Arguments

`d` — Bloomberg data
structure (default) | table | timetable

Bloomberg data, returned as a structure, table, or timetable. The data type of the Bloomberg data depends on the DataReturnFormat and DatetimeType properties of the connection object. For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`sec` — Security list
cell array of character vectors

Security list, returned as a cell array of character vectors for the corresponding securities in `s`. The contents of `sec` are identical in value and order to `s`. You can return securities with any of the following identifiers:

- `buid`
- `cats`
- `cins`
- `common`
- `cusip`
- `isin`
- `sedol1`
- `sedol2`
- `sicovam`
- `svm`
- `ticker` (default)
- `wpk`

Tips

- Bloomberg V3 data supports additional name-value pair arguments. To access further information on these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.
- You can check data and field availability by using the Bloomberg Excel® Add-In.
- For a Bloomberg B-PIPE connection, `d` returns an additional field named `EID`. `EID` means entitlement identifier. For details, see the *Bloomberg API Developer's Guide*.

Version History

Introduced in R2010a

See Also

`blp` | `history` | `realtime` | `timeseries` | `close`

Topics

“Retrieve Bloomberg Current Data” on page 3-5
“Retrieve Current and Historical Data Using Bloomberg” on page 1-7
“Workflow for Bloomberg” on page 3-15

history

Historical data for Bloomberg connection V3

Syntax

```
d = history(c,s,f,fromdate,todate)
d = history(c,s,f,fromdate,todate,period)
d = history(c,s,f,fromdate,todate,period,currency)
d = history(c,s,f,fromdate,todate,period,currency,Name,Value)
[d,sec] = history( ___ )
```

Description

`d = history(c,s,f,fromdate,todate)` returns the historical data for the security list `s` and the connection object `c` for the fields `f` for the dates `fromdate` through `todate`. Date strings can be input in any format recognized by MATLAB. `sec` is the security list that maps the order of the return data. The return data `d` is sorted to match the input order of `s`.

`d = history(c,s,f,fromdate,todate,period)` returns the historical data for the fields `f` and the dates `fromdate` through `todate` with a specific periodicity `period`.

`d = history(c,s,f,fromdate,todate,period,currency)` returns the historical data for the security list `s` for the fields `f` and the dates `fromdate` through `todate` based on the given currency `currency`.

`d = history(c,s,f,fromdate,todate,period,currency,Name,Value)` returns the historical data for the security list `s` using additional options specified by one or more name-value pair arguments.

`[d,sec] = history( ___ )` additionally returns the security list `sec` using any of the input argument combinations in the previous syntaxes. The return data, `d` and `sec`, are sorted to match the input order of `s`.

Examples

Daily Closing Prices Within Date Range

First, create a Bloomberg Desktop connection. Then, retrieve the daily closing price for a security within a date range.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Get the daily closing price from August 1, 2010, through August 10, 2010, for the IBM security.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','8/10/2010')

d =

     734352.00        123.55
     734353.00        123.18
     734354.00        124.03
     734355.00        124.56
     734356.00        123.58
     734359.00        125.34
     734360.00        125.19


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Monthly Closing Prices Within Date Range

First, create a Bloomberg Desktop connection. Then, retrieve the monthly closing prices for a security within a date range.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Get the monthly closing price from August 1, 2010, through December 10, 2010, for the IBM security.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','12/10/2010','monthly')

d =

     734360.00        125.19
     734391.00        121.53
     734421.00        131.85
     734452.00        139.78
     734482.00        138.13


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Monthly Closing Prices Within Date Range Using US Currency

First, create a Bloomberg Desktop connection. Then, retrieve the monthly closing prices for a security within a date range. Specify prices using the US currency.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Get the monthly closing price from August 1, 2010, through December 10, 2010, for the IBM security in US currency `'USD'`.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','12/10/2010','monthly','USD')
```

```
d =

     734360.00        125.19
     734391.00        121.53
     734421.00        131.85
     734452.00        139.78
     734482.00        138.13


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Monthly Closing Prices Within Date Range Using Currency with Specified Period

First, create a Bloomberg Desktop connection. Then, retrieve the monthly closing prices for a security within a date range. Specify prices using the US currency. Specify period values to customize the returned data.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Get the monthly closing price from August 1, 2010, through August 1, 2011, for the IBM security in US currency. The period values `'monthly'`, `'actual'`, and `'all_calendar_days'` specify returning actual monthly data for all calendar days. The period value `'nil_value'` specifies filling missing data values with a `NaN`.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','8/01/2011',{'monthly','actual',...
                  'all_calendar_days','nil_value'},'USD')
```

```
d =

     734351.00        128.40
     734382.00        125.77
     734412.00        135.64
     734443.00        143.32
     734473.00        144.41
     734504.00        146.76
     734535.00        163.56
     734563.00        159.97
     734594.00        164.27
     734624.00        170.58
     734655.00        166.56
     734685.00        174.54
     734716.00        180.75


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Daily Closing Prices Within Date Range Using Currency with Name-Value Pairs

First, create a Bloomberg Desktop connection. Then, retrieve the daily closing prices for a security within a date range. Specify prices using the US currency. Use name-value pair arguments to adjust the prices.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Get the daily closing price from August 1, 2010, through August 10, 2010, for the IBM security in U.S. currency `'USD'`. The prices are adjusted for normal cash and splits.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','8/10/2010','daily','USD',...
```

```
                'adjustmentNormal',true,...
                'adjustmentSplit',true)

d =

     734352.00        123.55
     734353.00        123.18
     734354.00        124.03
     734355.00        124.56
     734356.00        123.58
     734359.00        125.34
     734360.00        125.19


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Daily Closing Prices Using CUSIP Number and Pricing Source

First, create a Bloomberg Desktop connection. Then, retrieve the daily closing prices for a security within a date range. Specify the security using the CUSIP number and a pricing source.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Get the daily closing price from January 1, 2012, through January 1, 2013, for the security specified with a CUSIP number `/cusip/459200101` and with pricing source `BGN`.

`d` contains the numeric representation for the date in the first column and the closing price in the second column.

```
d = history(c,'/cusip/459200101@BGN','LAST_PRICE',...
            '01/01/2012','01/01/2013')

d =

     734871.00        180.69
     734872.00        179.96
     734873.00        179.10
     ...
```

Close the Bloomberg connection.

```
close(c)
```

Closing Prices Within Date Range Using International Date Format

First, create a Bloomberg Desktop connection. Then, retrieve the closing prices for a security within a date range. Specify the dates for the range using an international date format.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Return the closing price for the given dates in international format for the security `'MSFT@BGN US Equity'`.

```
stDt = datetime('01/06/11','InputFormat','dd/MM/yy');
endDt = datetime('01/06/12','InputFormat','dd/MM/yy');
[d,sec] = history(c,'MSFT@BGN US Equity','LAST_PRICE',...
                  stDt,endDt,{'previous_value','all_calendar_days'})
```

```
d =

     734655.00         22.92
     734656.00         22.72
     734657.00         22.42
     ...

sec =

    'MSFT@BGN US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Median Estimated Earnings Per Share Using Override Fields

First, create a Bloomberg Desktop connection. Then, retrieve the median earnings per share for a security within a date range. Specify an override field and value.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the median estimated earnings per share for AkzoNobel® from October 1, 2010, through October 30, 2010. When specifying Bloomberg override fields, use the character vector

'overrideFields'. The overrideFields argument must be an n-by-2 cell array, where the first column is the override field and the second column is the override value.

```
d = history(c,'AKZA NA Equity','BEST_EPS_MEDIAN', ...
    datetime('01.10.2010','InputFormat','dd.MM.yyyy'), ...
    datetime('30.10.2010','InputFormat','dd.MM.yyyy'), ...
    {'daily','calendar'},[],'overrideFields', ...
    {'BEST_FPERIOD_OVERRIDE','BF'})
```

```
d =

     734412.00          3.75
     734415.00          3.75
     734416.00          3.75
     ...
```

d returns the numeric representation for the date in the first column and the median estimated earnings per share in the second column.

Close the Bloomberg connection.

```
close(c)
```

Historical Data as Table with Dates

Create a Bloomberg® connection, and then retrieve closing prices for a historical date range. The history function returns data for dates as a datetime array.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using blpsrv or Bloomberg B-PIPE® using bpipe.

Return data as a table by setting the DataReturnFormat property of the connection object. If you do not set this property, the history function returns data as a numeric array.

Return dates as a datetime array by setting the DatetimeType property of the connection object. In this case, the table contains dates in variables that are datetime arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve historical closing prices for IBM® from August 1, 2010 through August 10, 2010. d is a table that contains dates as a datetime array.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE', ...
    '8/01/2010','8/10/2010')
```

```
d =
```

```
  7×2 table

      DATE        LAST_PRICE
    ___________    __________

    02-Aug-2010      130.76
    03-Aug-2010      130.37
    04-Aug-2010      131.27
    05-Aug-2010      131.83
    06-Aug-2010      130.14
    09-Aug-2010      132.00
    10-Aug-2010      131.84


sec =

  1×1 cell array

    {'IBM US Equity'}
```

Access dates in the returned data.

```
d.DATE
```

```
ans =

  7×1 datetime array

   02-Aug-2010
   03-Aug-2010
   04-Aug-2010
   05-Aug-2010
   06-Aug-2010
   09-Aug-2010
   10-Aug-2010
```

Close the Bloomberg connection.

```
close(c)
```

Historical Data as Timetable

Create a Bloomberg® connection, and then retrieve closing prices for a historical date range. The `history` function returns data as a `timetable`.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE® using `bpipe`.

Return data as a `timetable` by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `history` function returns data as a numeric array.

```
c.DataReturnFormat = 'timetable';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve historical closing prices for IBM® from August 1, 2010 through August 10, 2010. `d` is a `timetable` that contains dates in the first column.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE', ...
    '8/01/2010','8/10/2010')
```

```
d =

  7×1 timetable

       DATE        LAST_PRICE
    ____________    __________

    02-Aug-2010       130.76
    03-Aug-2010       130.37
    04-Aug-2010       131.27
    05-Aug-2010       131.83
    06-Aug-2010       130.14
    09-Aug-2010       132.00
    10-Aug-2010       131.84


sec =

  1×1 cell array

    {'IBM US Equity'}
```

Access dates in the returned data.

```
d.DATE
```

```
ans =

  7×1 datetime array

   02-Aug-2010
   03-Aug-2010
   04-Aug-2010
   05-Aug-2010
   06-Aug-2010
   09-Aug-2010
   10-Aug-2010
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

f — Bloomberg data fields

character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

period — Periodicity

`'daily'` | `'weekly'` | `'monthly'` | `'quarterly'` | ...

Periodicity, specified as one of these values to denote the data to return. For specifying multiple values, use a cell array. For example, when `period` is set to `{'daily','all_calendar_days'}`, `history` returns daily data for all calendar days, and reports missing data as `NaN`s. When `period` is set to `'active_days_only'`, `history` returns data using the default periodicity for active trading days only. The default periodicity depends on the security. If a security is reported on a monthly basis, the default periodicity is monthly. These tables show the values for `period`.

To specify the periodicity of the return data, see this table.

Value	Description
`'daily'`	Return data for each day.
`'weekly'`	Return data for each week.
`'monthly'`	Return data for each month.
`'quarterly'`	Return data for each quarter.
`'semi_annually'`	Return data semiannually.
`'yearly'`	Return data for each year.

The anchor date is the date to which all other reported dates are related. To specify the anchor date, see this table.

Value	Description
`'actual'`	Anchor date specification for an actual date. For this function, for periodicities other than daily, `todate` is the anchor date. If the period is weekly and the `todate` is a Thursday, every data point is a Thursday, or the nearest prior business day to Thursday. If the period is monthly and the `todate` is the 20th of a month, every data point is the 20th of each month in the date range.
`'calendar'`	Anchor date specification for a calendar year.
`'fiscal'`	Anchor date specification for a fiscal year.
`'none'`	Do not specify the anchor date.

To specify returning data for particular days, see this table.

Value	Description
`'non_trading_weekdays'`	Return data for all weekdays.
`'all_calendar_days'`	Return data for all calendar days.
`'active_days_only'`	Return data for only active trading days.

To specify how to fill missing values, see this table.

Value	Description
`'previous_value'`	Fill missing values with previous values for dates without trading activity for the security. If no previous value exists in the month before the `fromdate`, this function retains the missing values.
`'nil_value'`	Fill missing values with a `NaN` for dates without trading activity for the security.

Data Types: `char` | `cell`

`currency` — Currency
character vector | string scalar

Currency, specified as a character vector or string scalar to denote the ISO® code for the currency of the returned data. For example, to specify output money values in U.S. currency, use `USD` for this argument.

Data Types: `char` | `string`

`fromdate` — Beginning date
double scalar | character vector | string scalar | datetime

Beginning date for the historical data, specified as a double scalar, character vector, string scalar, or `datetime` array.

Data Types: `datetime` | `double` | `char` | `string`

`todate` — End date

double scalar | character vector | string scalar | datetime

End date for the historical data, specified as a double scalar, character vector, string scalar, or `datetime` array.

Data Types: `datetime` | `double` | `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'adjustmentNormal',true`

`overrideFields` — Override fields

cell array

Override fields, specified as the comma-separated pair consisting of `'overrideFields'` and an `n`-by-`2` cell array. The first column of the cell array is the override field and the second column is the override value.

Example: `'overrideFields',
{'IVOL_DELTA_LEVEL','DELTA_LVL_10';'IVOL_DELTA_PUT_OR_CALL','IVOL_PUT';'IVOL_MATURITY','MATURITY_1STM'}`

Data Types: `cell`

`adjustmentNormal` — Historical normal pricing adjustment

true | false

Historical normal pricing adjustment, specified as the comma-separated pair consisting of `'adjustmentNormal'` and a Boolean to reflect:

- Regular Cash
- Interim
- 1st Interim
- 2nd Interim
- 3rd Interim
- 4th Interim
- 5th Interim
- Income
- Estimated
- Partnership Distribution
- Final
- Interest on Capital
- Distribution
- Prorated

For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

`adjustmentAbnormal` — Historical abnormal pricing adjustment
true | false

Historical abnormal pricing adjustment, specified as the comma-separated pair consisting of `'adjustmentAbnormal'` and a Boolean to reflect:

- Special Cash
- Liquidation
- Capital Gains
- Long-Term Capital Gains
- Short-Term Capital Gains
- Memorial
- Return of Capital
- Rights Redemption
- Miscellaneous
- Return Premium
- Preferred Rights Redemption
- Proceeds/Rights
- Proceeds/Shares
- Proceeds/Warrants

For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

`adjustmentSplit` — Historical split pricing or volume adjustment
true | false

Historical split pricing or volume adjustment, specified as the comma-separated pair consisting of `'adjustmentSplit'` and a Boolean to reflect:

- Spin-Offs
- Stock Splits/Consolidations
- Stock Dividend/Bonus
- Rights Offerings/Entitlement

For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

`adjustmentFollowDPDF` — Historical pricing adjustment
true (default) | false

Historical pricing adjustment, specified as the comma-separated pair consisting of `'adjustmentFollowDPDF'` and a Boolean. Setting this name-value pair follows the **DPDF <GO>** option from the Bloomberg terminal. For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

Output Arguments

d — Bloomberg historical data

numeric array (default) | table | timetable

Bloomberg historical data, returned as a numeric array, table, or timetable. The data type of the historical data depends on the DataReturnFormat and DatetimeType properties of the connection object. The first column (or field) in the historical data contains the date. The remaining columns contain the requested data fields.

For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

sec — Security list

cell array of character vectors

Security list, returned as a cell array of character vectors for the corresponding securities in `s`. The contents of `sec` are identical in value and order to `s`. You can return securities with any of the following identifiers:

- `buid`
- `cats`
- `cins`
- `common`
- `cusip`
- `isin`
- `sedol1`
- `sedol2`
- `sicovam`
- `svm`
- `ticker` (default)
- `wpk`

Tips

- For better performance, add the Bloomberg file `blpapi3.jar` to the MATLAB static Java class path by modifying the file `$MATLAB/toolbox/local/javaclasspath.txt`. For details about the static Java class path, see "Static Path of Java Class Path".
- You can check data and field availability by using the Bloomberg Excel Add-In.

Version History

Introduced in R2010a

See Also

`blp` | `realtime` | `timeseries` | `getdata` | `close`

Topics

“Retrieve Bloomberg Historical Data” on page 3-7
“Retrieve Current and Historical Data Using Bloomberg” on page 1-7
“Workflow for Bloomberg” on page 3-15

isconnection

Determine Bloomberg connection V3

Syntax

```
v = isconnection(c)
```

Description

`v = isconnection(c)` returns `true` if `c` is a valid Bloomberg V3 connection and `false` otherwise.

Examples

Validate the Bloomberg Connection

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

Output Arguments

v — Valid Bloomberg connection

true | false

Valid Bloomberg connection, returned as a logical true, 1, or a logical false, 0.

Version History

Introduced in R2010b

See Also

`blp` | `blpsrv` | `bpipe` | `close` | `getdata`

Topics

“Connect to Bloomberg” on page 3-2
“Workflow for Bloomberg” on page 3-15

lookup

Find information about securities for Bloomberg connection V3

Syntax

```
l = lookup(c,q,reqtype,Name,Value)
```

Description

`l = lookup(c,q,reqtype,Name,Value)` retrieves data based on criteria in the query `q` for a specific request type `reqtype` using the Bloomberg connection `c`. For additional information about the query criteria and the possible name-value pair combinations, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Examples

Look Up Security

Create a Bloomberg® connection, and then use the Security Lookup to retrieve information about the IBM® corporate bond. For details about Bloomberg and the parameter values you can set, see the *Bloomberg API Developer's Guide* using the **WAPI<GO>** option from the Bloomberg terminal.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE® using `bpipe`.

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `lookup` function returns data as a structure.

```
c.DataReturnFormat = 'table';
```

Retrieve the instrument data for an IBM corporate bond with a maximum of 20 rows of data. The Security Lookup returns the security names and descriptions.

```
insts = lookup(c,'IBM','instrumentListRequest','maxResults',20, ...
    'yellowKeyFilter','YK_FILTER_CORP', ...
    'languageOverride','LANG_OVERRIDE_NONE');
```

Display the first three rows in the table. The first column contains the IBM corporate bond names, and the second column contains the bond descriptions.

```
insts(1:3,:)
```

```
ans =

  3×2 table
```

```
        security                    description
    ____________________    ____________________________________

    'DD103619 <corp>'    'International Business Machines Corp'
    '459200AG <corp>'    'International Business Machines Corp'
    'EC767659 <corp>'    'International Business Machines Corp'
```

Close the Bloomberg connection.

```
close(c)
```

Look Up Curve

Use the Curve Lookup to retrieve information about the `'GOLD'` related curve `'CD1016'`. For details about Bloomberg and the parameter values you can set, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Connect to Bloomberg.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the curve data for the credit default swap subtype of corporate bonds for a `'GOLD'` related curve `'CD1016'`. Return a maximum of `10` rows of data for the U.S. with `'USD'` currency.

```
curves = lookup(c,'GOLD','curveListRequest','maxResults',10,...
                'countryCode','US','currencyCode','USD',...
                'curveid','CD1016','type','CORP','subtype','CDS')
```

```
curves =

            curve: {'YCCD1016 Index'}
      description: {'Goldman Sachs Group Inc/The'}
          country: {'US'}
         currency: {'USD'}
          curveid: {'CD1016'}
             type: {'CORP'}
          subtype: {'CDS'}
        publisher: {'Bloomberg'}
            bbgid: {''}
```

One row of data displays as Bloomberg curve name `'YCCD1016 Index'` with Bloomberg description `'Goldman Sachs Group Inc/The'` in the U.S. with `'USD'` currency. The Bloomberg short-form identifier for the curve is `'CD1016'`. Bloomberg is the publisher and the `bbgid` is blank.

Close the Bloomberg connection.

```
close(c)
```

Look Up Government Security

Use the Government Security Lookup to retrieve information for United States Treasury bonds. For details about Bloomberg and the parameter values you can set, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Connect to Bloomberg.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Filter government security data with ticker filter of `'T'` for a maximum of `10` rows of data.

```
govts = lookup(c,'T','govtListRequest','maxResults',10,...
               'partialMatch',false)
```

```
govts =

      parseky: {10x1 cell}
         name: {10x1 cell}
       ticker: {10x1 cell}
```

The Government Security Lookup returns `parseky` data, the name, and ticker of the United States Treasury bonds.

Display the `parseky` data.

```
govts.parseky
```

```
ans =
    '912828VS Govt'
    '912828RE Govt'
    '912810RC Govt'
    '912810RB Govt'
    '912828VU Govt'
    '912828VV Govt'
    '912828VB Govt'
    '912828VR Govt'
    '912828VW Govt'
    '912828VQ Govt'
```

Display the names of the United States Treasury bonds.

```
govts.name
```

```
ans =
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
```

Display the tickers of the United States Treasury bonds.

```
govts.ticker
```

```
ans =
     'T'
     'T'
     'T'
     'T'
     'T'
     'T'
     'T'
     'T'
     'T'
     'T'
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

q — Keyword query

character vector | string scalar | cell array of character vectors | string array

Keyword query, specified as a character vector, string scalar, cell array of character vectors, or string array. Each character vector or string denotes an item for which information is requested. For example, the keyword query can be a security, a curve type, or a filter ticker.

Data Types: `char` | `cell` | `string`

reqtype — Request type

`'instrumentListRequest'` | `'curveListRequest'` | `'govtListRequest'`

Request type, specified as the preceding values to denote the type of information request. `'instrumentListRequest'` denotes a security or instrument lookup request. `'curveListRequest'` denotes a curve lookup request. `'govtListRequest'` denotes a government lookup request for government securities.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'maxResults'`, `20`, `'yellowKeyFilter'`, `'YK_FILTER_CORP'`, `'languageOverride'`, `'LANG_OVERRIDE_NONE'`, `'countryCode'`, `'US'`, `'currencyCode'`, `'USD'`, `'curveid'`, `'CD1016'`, `'type'`, `'CORP'`, `'subtype'`, `'CDS'`, `'partialMatch'`, `false`

`maxResults` — Number of rows in result data
numeric scalar

Number of rows in the result data, specified as the comma-separated pair consisting of `'maxResults'` and a numeric scalar to denote the total maximum number of rows of information to return. Result data can be one or more rows of data no greater than the number specified.

Data Types: `double`

`yellowKeyFilter` — Bloomberg yellow key filter
character vector | string scalar

Bloomberg yellow key filter, specified as the comma-separated pair consisting of `'yellowKeyFilter'` and a unique character vector or string scalar to denote the particular yellow key for government securities, corporate bonds, equities, and commodities, for example.

Data Types: `char` | `string`

`languageOverride` — Language override
character vector | string scalar

Language override, specified as the comma-separated pair consisting of `'languageOverride'` and a unique character vector or string scalar to denote a translation language for the result data.

Data Types: `char` | `string`

`countryCode` — Country code
character vector | string scalar

Country code, specified as the comma-separated pair consisting of `'countryCode'` and a character vector or string scalar to denote the country for the result data.

Data Types: `char` | `string`

`currencyCode` — Currency code
character vector | string scalar

Currency code, specified as the comma-separated pair consisting of `'currencyCode'` and a character vector or string scalar to denote the currency for the result data.

Data Types: `char` | `string`

`curveID` — Bloomberg short-form identifier for curve
character vector | string scalar

Bloomberg short-form identifier for a curve, specified as the comma-separated pair consisting of `'curveID'` and a character vector or string scalar.

Data Types: `char` | `string`

`type` — Bloomberg market sector type
character vector | string scalar

Bloomberg market sector type corresponding to the Bloomberg yellow keys, specified as the comma-separated pair consisting of `'type'` and a character vector or string scalar.

Data Types: `char` | `string`

`subtype` — Bloomberg market sector subtype
character vector | string scalar

Bloomberg market sector subtype, specified as the comma-separated pair consisting of `'subtype'` and a character vector or string scalar to further delineate the market sector type.

Data Types: `char` | `string`

`partialMatch` — Partial match on ticker
`true` | `false`

Partial match on ticker, specified as the comma-separated pair consisting of `'partialMatch'` and `true` or `false`. When set to `true`, you can filter securities by setting `q` to a query such as `'T*'`. When set to `false`, the securities are unfiltered.

Data Types: `logical`

Output Arguments

`l` — Lookup information
structure (default) | table

Lookup information, returned as a structure or table containing set properties depending on the request type. The data type of the lookup information depends on the DataReturnFormat property of the connection object.

For a list of the set properties and their descriptions, see the following tables.

'instrumentListRequest' Properties

Property	Description
`security`	Security name
`description`	Security long name

'curveListRequest' Properties

Property	Description
`curve`	Bloomberg curve name
`description`	Bloomberg description
`country`	Country code
`currency`	Currency code
`curveid`	Bloomberg short-form identifier for the curve
`type`	Bloomberg market sector type
`subtype`	Bloomberg market sector subtype
`publisher`	Bloomberg specified as publisher
`bbgid`	Bloomberg identifier

'govtListRequest' Properties

Property	Description
`parsekey`	Bloomberg security identifier (ticker or CUSIP, for example), price source, and source key (Bloomberg yellow key)
`name`	Government security name
`ticker`	Government security ticker

Version History

Introduced in R2014a

See Also

`blp` | `getdata` | `history` | `realtime` | `timeseries` | `close`

Topics

“Retrieve Bloomberg Current Data” on page 3-5
“Retrieve Current and Historical Data Using Bloomberg” on page 1-7
“Workflow for Bloomberg” on page 3-15

portfolio

Current portfolio data for Bloomberg connection V3

Syntax

```
d = portfolio(c,p,f)
d = portfolio(c,p,f,o,ov)
[d,plist] = portfolio( ___ )
```

Description

`d = portfolio(c,p,f)` returns current portfolio data for the fields `f` in the portfolio `p` using the Bloomberg connection `c`.

`d = portfolio(c,p,f,o,ov)` returns current portfolio data using override field `o` and override value `ov`.

`[d,plist] = portfolio( ___ )` also returns the portfolio list `plist` using any of the input argument combinations in the previous syntaxes.

Examples

Request Portfolio Data

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Request portfolio data for a custom portfolio with portfolio identifier `U335877-1 Client`. Request data using all fields `f`.

```
p = 'U335877-1 Client';
f = {'PORTFOLIO_MEMBERS','PORTFOLIO_MPOSITION',...
     'PORTFOLIO_MWEIGHT','PORTFOLIO_DATA'};

d = portfolio(c,p,f)
```

```
d =

    PORTFOLIO_MPOSITION: {{0x1 cell}}
      PORTFOLIO_MWEIGHT: {{0x1 cell}}
         PORTFOLIO_DATA: {{0x1 cell}}
      PORTFOLIO_MEMBERS: {{0x1 cell}}
```

`d` is a structure that contains portfolio data. Each structure field corresponds to data for each portfolio field.

Close the connection.

```
close(c)
```

Request Portfolio Data Using Specific Date

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Request portfolio data for a custom portfolio with portfolio identifier `U335877-1 Client`. Request data using all fields `f`. Filter the portfolio data by specifying the date of November 3, 2014, using the override value `REFERENCE_DATE` equal to `20141103`.

```
p = 'U335877-1 Client';
f = {'PORTFOLIO_MEMBERS','PORTFOLIO_MPOSITION',...
     'PORTFOLIO_MWEIGHT','PORTFOLIO_DATA'};
o = {'REFERENCE_DATE'};
ov = {'20141103'};

[d,plist] = portfolio(c,p,f,o,ov)

d =

    PORTFOLIO_MPOSITION: {{0x1 cell}}
      PORTFOLIO_MWEIGHT: {{0x1 cell}}
         PORTFOLIO_DATA: {{0x1 cell}}
      PORTFOLIO_MEMBERS: {{0x1 cell}}

plist =

    'U335877-1 Client'
```

`d` is a structure that contains portfolio data. Each structure field corresponds to data for each portfolio field.

`plist` is a cell array that contains the portfolio identifier.

Close the connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

p — Portfolio

character vector | string scalar

Portfolio, specified as a character vector or string scalar. Specify the portfolio by the ID that you can find in the upper-right corner of the portfolio display page. Append the text `' Client'` (without quotes) to the ID. For example, if the ID is `U335877-1`, then specify `'U335877-1 Client'`.

Access the portfolio display page by using the **PRTU<GO>** option from the Bloomberg terminal. For details, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `'U335877-1 Client'`

Data Types: `char` | `cell` | `string`

f — Portfolio fields
`'PORTFOLIO_DATA'` | `'PORTFOLIO_MEMBERS'` | `'PORTFOLIO_MPOSITION'` | `'PORTFOLIO_MWEIGHT'`

Portfolio fields, specified as one of the preceding values for one field. To specify multiple fields, use a cell array of these values.

Bloomberg Field Name	Bloomberg Field Description
`'PORTFOLIO_DATA'`	Returns a list of the identifiers, positions, market values, cost, cost date, and cost foreign exchange rate of each security in a custom portfolio.
`'PORTFOLIO_MEMBERS'`	Returns a list of identifiers for the members of a custom portfolio.
`'PORTFOLIO_MPOSITION'`	Returns a list of identifiers and the position for each security in a custom portfolio.
`'PORTFOLIO_MWEIGHT'`	Returns a list of identifiers and the percentage weight for each security in a custom portfolio.

Data Types: `char` | `cell`

o — Bloomberg override field
character vector | string scalar | cell array of character vectors | string array

Bloomberg override field, specified as a character vector, string scalar, cell array of character vectors, or string array. The Bloomberg value `'REFERENCE_DATE'` denotes returning Bloomberg data for a specific date.

Data Types: `char` | `cell` | `string`

ov — Bloomberg override field value
`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field value, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field value. A cell array of character vectors or string array denotes multiple Bloomberg override field values. Use this field value to filter the Bloomberg data result set.

Example: `'20100101'`

Data Types: `char` | `cell` | `string`

Output Arguments

`d` — Portfolio data
structure (default) | table

Portfolio data, returned as a structure or table. The data type of the portfolio data depends on the DataReturnFormat property of the connection object.

`plist` — Portfolio list
cell array of character vectors

Portfolio list, returned as a cell array of character vectors for the corresponding portfolio identifiers in `p`. The contents of `plist` are identical in value and order to `p`.

Version History

Introduced in R2015b

See Also

`blp` | `getdata` | `history` | `realtime` | `timeseries` | `close`

Topics

“Retrieve Bloomberg Current Data” on page 3-5
“Workflow for Bloomberg” on page 3-15

realtime

Real-time data for Bloomberg connection V3

Syntax

```
d = realtime(c,s,f)
[subs,t] = realtime(c,s,f,eventhandler)
```

Description

`d = realtime(c,s,f)` returns the data for the given connection `c`, security list `s`, and requested fields `f`. `realtime` accesses the Bloomberg Market Data service.

`[subs,t] = realtime(c,s,f,eventhandler)` returns the subscription list `subs` and the timer `t` associated with the real-time event handler for the subscription list. Given connection `c`, the `realtime` function subscribes to a security or securities `s` and requests fields `f`, to update in real time while running an event handler `eventhandler`.

Examples

Retrieve Data for One Security

Retrieve a snapshot of data for one security only.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the last trade and volume of the IBM security.

```
d = realtime(c,'IBM US Equity',{'Last_Trade','Volume'})
```

```
d =

    LAST_TRADE: '181.76'
        VOLUME: '7277793'
```

Close the Bloomberg connection.

```
close(c)
```

Retrieve Data for One Security Using the Event Handler `v3stockticker`

You can create your own event handler function to process Bloomberg data. For this example, use the event handler `v3stockticker` that returns Bloomberg stock tick data.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the last trade and volume for the IBM security using the event handler `v3stockticker`.

`v3stockticker` requires the input argument `f` of `realtime` to be `'Last_Trade'` , `'Volume'`, or both.

```
[subs,t] = realtime(c,'IBM US Equity',{'Last_Trade','Volume'},...
                    'v3stockticker')
```

```
subs =

com.bloomberglp.blpapi.SubscriptionList@79f07684

   Timer Object: timer-2

   Timer Settings
      ExecutionMode: fixedRate
             Period: 0.05
          BusyMode: drop
           Running: on

   Callbacks
          TimerFcn: 1x4 cell array
          ErrorFcn: ''
          StartFcn: ''
           StopFcn: ''

** IBM US Equity ** 100 @ 181.81 29-Oct-2013 15:48:50
** IBM US Equity ** 100 @ 181.795 29-Oct-2013 15:48:50
** IBM US Equity ** 100 @ 181.8065 29-Oct-2013 15:48:51
...
```

`realtime` returns the Bloomberg subscription list object `subs` and the MATLAB timer object with its properties. Then, `realtime` returns the stock tick data for the IBM security with the volume and last trade price.

Real-time data continues to display until you execute the `stop` or `close` function.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Data for Multiple Securities Using the Event Handler `v3stockticker`

You can create your own event handler function to process Bloomberg data. For this example, use the event handler `v3stockticker` that returns Bloomberg stock tick data.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the last trade and volume for IBM and Ford Motor Company securities.

`v3stockticker` requires the input argument `f` of the `realtime` function to be `'Last_Trade'`, `'Volume'`, or both.

```
[subs,t] = realtime(c,{'IBM US Equity','F US Equity'}, ...
                    {'Last_Trade','Volume'},'v3stockticker')
```

```
subs =

com.bloomberglp.blpapi.SubscriptionList@6c1066f6


   Timer Object: timer-3

   Timer Settings
      ExecutionMode: fixedRate
             Period: 0.05
          BusyMode: drop
           Running: on

   Callbacks
          TimerFcn: 1x4 cell array
          ErrorFcn: ''
          StartFcn: ''
           StopFcn: ''

** IBM US Equity ** 32433 @ 181.85 29-Oct-2013 15:50:05
** IBM US Equity ** 200 @ 181.85 29-Oct-2013 15:50:05
** IBM US Equity ** 100 @ 181.86 29-Oct-2013 15:50:05
** F US Equity ** 300 @ 17.575 30-Oct-2013 10:14:06
** F US Equity ** 100 @ 17.57 30-Oct-2013 10:14:06
** F US Equity ** 100 @ 17.5725 30-Oct-2013 10:14:06
...
```

`realtime` returns the Bloomberg subscription list object `subs` and the MATLAB timer object with its properties. Then, `realtime` returns the stock tick data for the IBM and Ford Motor Company securities with the last trade price and volume.

Real-time data continues to display until you use the `stop` or `close` function.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Data for One Security Using the Event Handler `v3showtrades`

You can create your own event handler function to process Bloomberg data. For this example, use the event handler `v3showtrades` that creates a figure showing requested data for a security.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

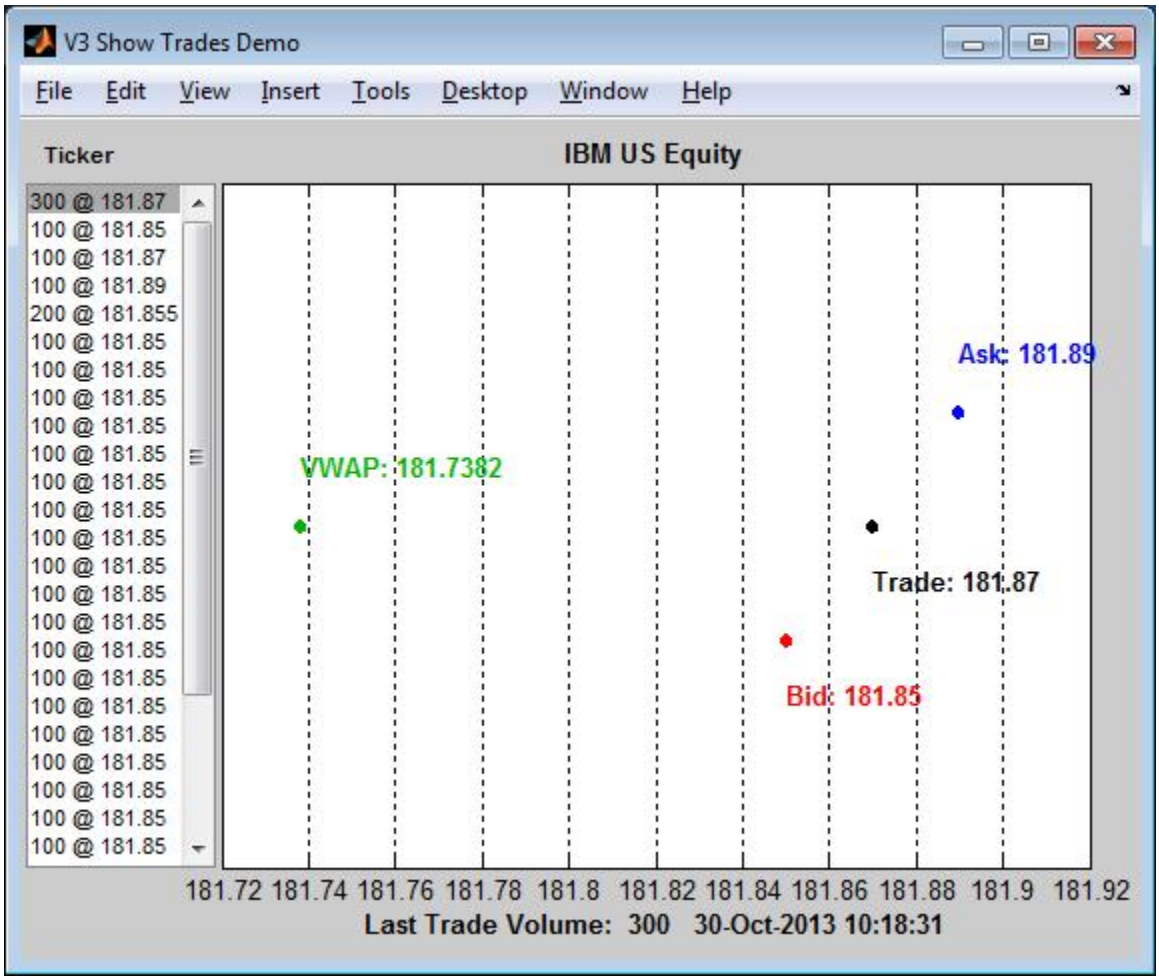
Retrieve volume, last trade, bid, ask, and volume weight adjusted price (VWAP) data for the IBM security using the event handler `v3showtrades`.

`v3showtrades` requires the input argument `f` of `realtime` to be any combination of: `'Last_Trade'`, `'Bid'`, `'Ask'`, `'Volume'`, and `'VWAP'`.

```
[subs,t] = realtime(c,'IBM US Equity',...
                    {'Last_Trade','Bid','Ask','Volume','VWAP'},...
                    'v3showtrades')
```

```
subs =

com.bloomberglp.blpapi.SubscriptionList@5c17dcdb


   Timer Object: timer-4

   Timer Settings
      ExecutionMode: fixedRate
             Period: 0.05
          BusyMode: drop
           Running: on

   Callbacks
          TimerFcn: 1x4 cell array
          ErrorFcn: ''
          StartFcn: ''
           StopFcn: ''
```

`realtime` returns the Bloomberg subscription list object `subs` and the MATLAB timer object with its properties. Then, `v3showtrades` displays a figure showing volume, last trade, bid, ask, and volume weight adjusted price (VWAP) data for IBM.



Real-time data continues to display until you execute the `stop` or `close` function.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Data for One Security Using the Event Handler `v3pricevol`

You can create your own event handler function to process Bloomberg data. For this example, use the event handler `v3pricevol` that creates a figure showing last price and volume data for a security.

Create the Bloomberg connection.

```
c = blp;
```

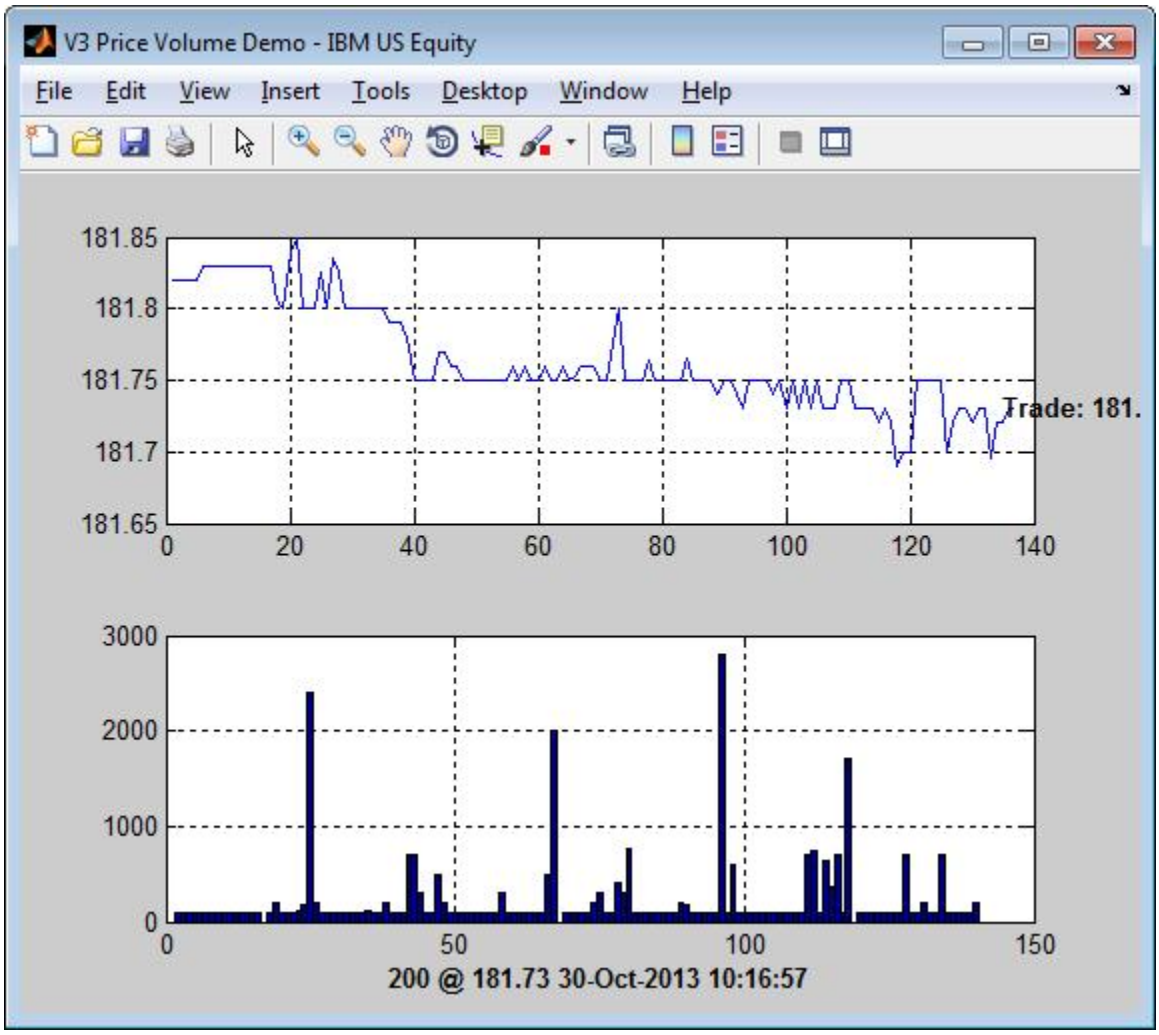
Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve last price and volume data for the IBM security using event handler `v3pricevol`.

`v3pricevol` requires the input argument `f` of `realtime` to be `'Last_Price'`, `'Volume'`, or both.

```
[subs,t] = realtime(c,'IBM US Equity',{'Last_Price','Volume'},...
                    'v3pricevol')

subs =

com.bloomberglp.blpapi.SubscriptionList@16f66676


   Timer Object: timer-5

   Timer Settings
      ExecutionMode: fixedRate
             Period: 0.05
          BusyMode: drop
           Running: on

   Callbacks
          TimerFcn: 1x4 cell array
          ErrorFcn: ''
          StartFcn: ''
           StopFcn: ''
```

`realtime` returns the Bloomberg subscription list object `subs` and the MATLAB timer object with its properties. Then, `v3pricevol` displays a figure showing last price and volume data for IBM.



Real-time data continues to display until you execute the `stop` or `close` function.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

`f` — Bloomberg data fields
character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

`eventhandler` — Event handler
character vector | string scalar

Event handler, specified as a character vector or string scalar that denotes the name of an event handler function that you define. You can define an event handler function to process any type of real-time Bloomberg events. The specified event handler function runs every time the timer fires.

Data Types: `char` | `string`

Output Arguments

`d` — Bloomberg data
structure (default) | table | timetable

Bloomberg data, returned as a structure, table, or timetable. The data type of the Bloomberg data depends on the DataReturnFormat and DatetimeType properties of the connection object. For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`subs` — Bloomberg subscription
object

Bloomberg subscription, returned as a Bloomberg object. For details about this object, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`t` — MATLAB timer
object

MATLAB timer, returned as a MATLAB object. For details about this object, see `timer`.

Version History

Introduced in R2010a

See Also

`blp` | `getdata` | `history` | `timeseries` | `stop` | `close`

Topics
"Retrieve Bloomberg Real-Time Data" on page 3-13

“Workflow for Bloomberg” on page 3-15
“Writing and Running Custom Event Handler Functions” on page 1-26

stop

Unsubscribe real-time requests for Bloomberg connection V3

Syntax

```
stop(c,subs,t)
stop(c,subs,[],s)
```

Description

`stop(c,subs,t)` unsubscribes real-time requests associated with the Bloomberg connection `c` and subscription list `subs`. `t` is the timer associated with the real-time callback for the subscription list.

`stop(c,subs,[],s)` unsubscribes real-time requests for each security `s` on the subscription list `subs`. The timer input is empty.

Examples

Stop Real-Time Requests

Unsubscribe to real-time data for one security.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the last trade and volume for the IBM security using the event handler `v3stockticker`.

`v3stockticker` requires the input argument `f` of `realtime` to be `'Last_Trade'`, `'Volume'`, or both.

```
[subs,t] = realtime(c,'IBM US Equity',{'Last_Trade','Volume'},...
                    'v3stockticker');
```

```
** IBM US Equity ** 100 @ 181.81 29-Oct-2013 15:48:50
** IBM US Equity ** 100 @ 181.795 29-Oct-2013 15:48:50
** IBM US Equity ** 100 @ 181.8065 29-Oct-2013 15:48:51
...
```

`realtime` returns the stock tick data for the IBM security with the volume and last trade price.

Stop the real-time data requests for the IBM security using the Bloomberg subscription `subs` and MATLAB timer object `t`.

```
stop(c,subs,t)
```

Close the Bloomberg connection.

```
close(c)
```

Stop Real-Time Requests for a Security List

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the last trade and volume for the security list `s` using the event handler `v3stockticker`. `s` contains securities for IBM, Google, and Ford Motor Company.

`v3stockticker` requires the input argument `f` of `realtime` to be `'Last_Trade'`, `'Volume'`, or both.

```
s = {'IBM US Equity','GOOG US Equity','F US Equity'};
[subs,t] = realtime(c,s,{'Last_Trade','Volume'},'v3stockticker');
```

```
** IBM US Equity ** 100 @ 181.81 29-Oct-2013 15:48:50
** IBM US Equity ** 100 @ 181.795 29-Oct-2013 15:48:50
** IBM US Equity ** 100 @ 181.8065 29-Oct-2013 15:48:51
...
```

`realtime` returns the stock tick data for the securities list `s` with the volume and last trade price.

Stop the real-time data requests for the securities list `s` using the Bloomberg subscription `subs`.

```
stop(c,subs,[],s)
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

`subs` — Bloomberg subscription

object

Bloomberg subscription, specified as a Bloomberg object. For details about this object, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`t` — MATLAB timer

object

MATLAB timer, specified as a MATLAB object. For details about this object, see `timer`.

`s` — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

Version History

Introduced in R2010a

See Also

`blp` | `getdata` | `history` | `realtime` | `timeseries` | `close`

Topics

“Retrieve Bloomberg Real-Time Data” on page 3-13
“Workflow for Bloomberg” on page 3-15

tahistory

Historical technical analysis for Bloomberg connection V3

Syntax

```
d = tahistory(c)
d = tahistory(c,s,startdate,enddate,study,period,Name,Value)
```

Description

`d = tahistory(c)` returns the Bloomberg V3 session technical analysis data study and element definitions.

`d = tahistory(c,s,startdate,enddate,study,period,Name,Value)` returns the Bloomberg V3 session technical analysis data study and element definitions with additional options specified by one or more name-value pair arguments.

Examples

Request Bloomberg Directional Movement Indicator (DMI) Study for Security

Return all available Bloomberg studies and use the DMI study to run a technical analysis for a security.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

List the available Bloomberg studies.

```
d = tahistory(c)
```

```
d =

          dmiStudyAttributes: [1x1 struct]
        smavgStudyAttributes: [1x1 struct]
         bollStudyAttributes: [1x1 struct]
          maoStudyAttributes: [1x1 struct]
           fgStudyAttributes: [1x1 struct]
          rsiStudyAttributes: [1x1 struct]
         macdStudyAttributes: [1x1 struct]
          tasStudyAttributes: [1x1 struct]
        emavgStudyAttributes: [1x1 struct]
       maxminStudyAttributes: [1x1 struct]
         ptpsStudyAttributes: [1x1 struct]
         cmciStudyAttributes: [1x1 struct]
         wlprStudyAttributes: [1x1 struct]
        wmavgStudyAttributes: [1x1 struct]
```

```
     trenderStudyAttributes: [1x1 struct]
         gocStudyAttributes: [1x1 struct]
        kltnStudyAttributes: [1x1 struct]
    momentumStudyAttributes: [1x1 struct]
         rocStudyAttributes: [1x1 struct]
         maeStudyAttributes: [1x1 struct]
       hurstStudyAttributes: [1x1 struct]
        chkoStudyAttributes: [1x1 struct]
          teStudyAttributes: [1x1 struct]
       vmavgStudyAttributes: [1x1 struct]
       tmavgStudyAttributes: [1x1 struct]
         atrStudyAttributes: [1x1 struct]
         rexStudyAttributes: [1x1 struct]
         adoStudyAttributes: [1x1 struct]
          alStudyAttributes: [1x1 struct]
         etdStudyAttributes: [1x1 struct]
         vatStudyAttributes: [1x1 struct]
        tvatStudyAttributes: [1x1 struct]
          pdStudyAttributes: [1x1 struct]
          rvStudyAttributes: [1x1 struct]
      ipmavgStudyAttributes: [1x1 struct]
       pivotStudyAttributes: [1x1 struct]
          orStudyAttributes: [1x1 struct]
         pcrStudyAttributes: [1x1 struct]
          bsStudyAttributes: [1x1 struct]
```

`d` contains structures pertaining to each available Bloomberg study.

Display the name-value pairs for the DMI study.

```
d.dmiStudyAttributes
```

```
ans =

               period: [1x104 char]
     priceSourceHigh: [1x123 char]
      priceSourceLow: [1x121 char]
    priceSourceClose: [1x125 char]
```

Obtain more information about the `period` property.

```
d.dmiStudyAttributes.period
```

```
ans =

DEFINITION period {

    Min Value = 1

    Max Value = 1

    TYPE Int64

} // End Definition: period
```

Run the DMI study for the IBM security for the last month with `period` equal to `14`, the high price, the low price, and the closing price.

```
d = tahistory(c,'IBM US Equity',floor(now)-30,floor(now),'dmi',...
              'all_calendar_days','period',14,...
```

```
            'priceSourceHigh','PX_HIGH',...
            'priceSourceLow','PX_LOW','priceSourceClose','PX_LAST')
```

```
d =

         date: [31x1 double]
     DMI_PLUS: [31x1 double]
    DMI_MINUS: [31x1 double]
          ADX: [31x1 double]
         ADXR: [31x1 double]
```

`d` contains a `studyDataTable` with one `studyDataRow` for each interval returned.

Display the first five dates in the returned data.

```
d.date(1:5,1)
```

```
ans =

     735507.00
     735508.00
     735509.00
     735510.00
     735511.00
```

Display the first five prices in the plus DI line.

```
d.DMI_PLUS(1:5,1)
```

```
ans =

         18.92
         17.84
         16.83
         15.86
         15.63
```

Display the first five prices in the minus DI line.

```
d.DMI_MINUS(1:5,1)
```

```
ans =

         30.88
         29.12
         28.16
         30.67
         29.24
```

Display the first five values of the Average Directional Index.

```
d.ADX(1:5,1)
```

```
ans =

         22.15
         22.28
         22.49
         23.15
         23.67
```

Display the first five values of the Average Directional Movement Index Rating.

```
d.ADXR(1:5,1)
```

```
ans =

         25.20
         25.06
         25.05
         25.60
         26.30
```

Close the Bloomberg connection.

```
close(c)
```

Request DMI Study for Security with Pricing Source

Run a technical analysis to return the DMI study for a security with a pricing source.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Run the DMI study for the Microsoft security with pricing source `ETPX` for the last month with `period` equal to `14`, the high price, the low price, and the closing price.

```
d = tahistory(c,'MSFT@ETPX US Equity',floor(now)-30,floor(now),...
              'dmi','all_calendar_days','period',14,...
              'priceSourceHigh','PX_HIGH','priceSourceLow','PX_LOW',...
              'priceSourceClose','PX_LAST')
```

```
d =

         date: [31x1 double]
     DMI_PLUS: [31x1 double]
    DMI_MINUS: [31x1 double]
          ADX: [31x1 double]
         ADXR: [31x1 double]
```

`d` contains a `studyDataTable` with one `studyDataRow` for each interval returned.

Display the first five dates in the returned data.

```
d.date(1:5,1)
```

```
ans =

     735507.00
     735508.00
     735509.00
     735510.00
     735511.00
```

Display the first five prices in the plus DI line.

```
d.DMI_PLUS(1:5,1)
```

```
ans =

         28.37
         30.63
         32.72
         30.65
         29.37
```

Display the first five prices in the minus DI line.

```
d.DMI_MINUS(1:5,1)
```

```
ans =

         21.97
         21.17
         19.47
         18.24
         17.48
```

Display the first values of the Average Directional Index.

```
d.ADX(1:5,1)
```

```
ans =

         13.53
         13.86
         14.69
         15.45
         16.16
```

Display the first five values of the Average Directional Movement Index Rating.

```
d.ADXR(1:5,1)
```

```
ans =

         15.45
         15.36
         15.53
         15.85
         16.37
```

Close the Bloomberg connection.

```
close(c)
```

Return DMI Study Data as Table with Dates

Create a Bloomberg® connection, and then return data for a DMI study. The `tahistory` function returns data for dates as a `datetime` array.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE® using `bpipe`.

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `tahistory` function returns data as a structure.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Run the DMI study for the IBM® security from June 12, 2017 through June 16, 2017 with `period` equal to 14, the high price, the low price, and the closing price.

```
d = tahistory(c,'IBM US Equity','6/12/2017','6/16/2017','dmi', ...
    'all_calendar_days','period',14,'priceSourceHigh','PX_HIGH', ...
    'priceSourceLow','PX_LOW','priceSourceClose','PX_LAST');
```

Access the DMI study data for the first three dates.

```
d(1:3,:)
```

```
ans =

  3×5 table

       date        DMI_PLUS    DMI_MINUS     ADX     ADXR
    ___________    ________    _________    _____    _____

    12-Jun-2017     30.48        16.31      33.93    45.26
    13-Jun-2017     28.88        15.45      33.67    44.10
    14-Jun-2017     26.62        18.98      32.46    42.67
```

`d` is a `table` that contains these columns:

- `date` -- Date
- `DMI_PLUS` -- Prices in plus DI line
- `DMI_MINUS` -- Prices in minus DI line
- `ADX` -- Average Directional Index values
- `ADXR` -- Average Directional Movement Index Rating values

Access the first three dates in the returned data.

```
d.date(1:3)
```

```
ans =
```

```
  3×1 datetime array

   12-Jun-2017
   13-Jun-2017
   14-Jun-2017
```

Close the Bloomberg connection.

```
close(c)
```

Return DMI Study Data as Timetable

Create a Bloomberg® connection, and then return data for a DMI study. The `tahistory` function returns data as a `timetable`.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE® using `bpipe`.

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `tahistory` function returns data as a structure.

```
c.DataReturnFormat = 'timetable';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Run the DMI study for the IBM® security from June 12, 2017 through June 16, 2017 with `period` equal to 14, the high price, the low price, and the closing price.

```
d = tahistory(c,'IBM US Equity','6/12/2017','6/16/2017','dmi', ...
    'all_calendar_days','period',14,'priceSourceHigh','PX_HIGH', ...
    'priceSourceLow','PX_LOW','priceSourceClose','PX_LAST');
```

Access the DMI study data for the first three dates.

```
d(1:3,:)
```

```
ans =

  3×4 timetable

      date        DMI_PLUS    DMI_MINUS     ADX     ADXR
   ___________    ________    _________    _____    _____

   12-Jun-2017     30.48        16.31      33.93    45.26
   13-Jun-2017     28.88        15.45      33.67    44.10
   14-Jun-2017     26.62        18.98      32.46    42.67
```

`d` is a `timetable` that contains these columns:

- `date` -- Date
- `DMI_PLUS` -- Prices in plus DI line
- `DMI_MINUS` -- Prices in minus DI line
- `ADX` -- Average Directional Index values
- `ADXR` -- Average Directional Movement Index Rating values

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

`s` — Security

character vector | string scalar

Security, specified as a character vector or string scalar for a single Bloomberg security.

Data Types: `char` | `string`

`startdate` — Start date

numeric scalar | character vector | string scalar

Start date, specified as a numeric scalar, character vector, or string scalar to denote the start date of the date range for the returned tick data.

Example: `floor(now-1)`

Data Types: `double` | `char` | `string`

`enddate` — End date

numeric scalar | character vector | string scalar

End date, specified as a numeric scalar, character vector, or string scalar to denote the end date of the date range for the returned tick data.

Example: `floor(now)`

Data Types: `double` | `char` | `string`

`study` — Study type

character vector | string scalar

Study type, specified as a character vector or string scalar to denote the study to use for historical analysis.

Data Types: `char` | `string`

`period` — Periodicity

`'daily'` | `'weekly'` | `'monthly'` | `'quarterly'` | ...

Periodicity, specified as one of these values to denote the data to return. For specifying multiple values, use a cell array. For example, when `period` is set to `{'daily','all_calendar_days'}`,

`tahistory` returns daily data for all calendar days, and reports missing data as `NaN`s. When `period` is set to `'active_days_only'`, `tahistory` returns data using the default periodicity for active trading days only. The default periodicity depends on the security. If a security is reported on a monthly basis, the default periodicity is monthly. These tables show the values for `period`.

To specify the periodicity of the return data, see this table.

Value	Description
`'daily'`	Return data for each day.
`'weekly'`	Return data for each week.
`'monthly'`	Return data for each month.
`'quarterly'`	Return data for each quarter.
`'semi_annually'`	Return data semiannually.
`'yearly'`	Return data for each year.

The anchor date is the date to which all other reported dates are related. To specify the anchor date, see this table.

Value	Description
`'actual'`	Anchor date specification for an actual date. For this function, for periodicities other than daily, `enddate` is the anchor date. If the period is weekly and the `enddate` is a Thursday, every data point is a Thursday, or the nearest prior business day to Thursday. If the period is monthly and the `enddate` is the 20th of a month, every data point is the 20th of each month in the date range.
`'calendar'`	Anchor date specification for a calendar year.
`'fiscal'`	Anchor date specification for a fiscal year.

To specify returning data for particular days, see this table.

Value	Description
`'non_trading_weekdays'`	Return data for all weekdays.
`'all_calendar_days'`	Return data for all calendar days.
`'active_days_only'`	Return data for only active trading days.

To specify how to fill missing values, see this table.

Value	Description
`'previous_value'`	Fill missing values with previous values for dates without trading activity for the security.
`'nil_value'`	Fill missing values with a `NaN` for dates without trading activity for the security.

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'period',14`, `'priceSourceHigh','PX_HIGH'`, `'priceSourceLow','PX_LOW'`, `'priceSourceClose','PX_LAST'`

Note For details about the full list of name-value pair arguments, see the Bloomberg tool located at `C:\blp\API\APIv3\bin\BBAPIDemo.exe`.

`period` — Period
numeric scalar

Period, specified as the comma-separated pair consisting of `'period'` and a numeric scalar. For details about the period, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `double`

`priceSourceHigh` — High price
character vector | string scalar

High price, specified as the comma-separated pair consisting of `'priceSourceHigh'` and a character vector or string scalar. For details about the high price, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

`priceSourceLow` — Low price
character vector | string scalar

Low price, specified as the comma-separated pair consisting of `'priceSourceLow'` and a character vector or string scalar. For details about the low price, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

`priceSourceClose` — Closing price
character vector | string scalar

Closing price, specified as the comma-separated pair consisting of `'priceSourceClose'` and a character vector or string scalar. For details about the closing price, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

Output Arguments

`d` — Technical analysis data
structure (default) | table | timetable

Technical analysis data, returned as a structure, table, or timetable. The data type of the returned data depends on the DataReturnFormat and DatetimeType properties of the connection object.

For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Version History

Introduced in R2012b

See Also

`blp` | `getdata` | `history` | `realtime` | `timeseries` | `close`

Topics

"Retrieve Bloomberg Current Data" on page 3-5
"Retrieve Current and Historical Data Using Bloomberg" on page 1-7
"Workflow for Bloomberg" on page 3-15

timeseries

Intraday tick data for Bloomberg connection V3

Syntax

```
d = timeseries(c,s,date)
d = timeseries(c,s,date,interval,field)
d = timeseries(c,s,date,[],field,options,values)

d = timeseries(c,s,{startdate,enddate})
d = timeseries(c,s,{startdate,enddate},interval,field)
d = timeseries(c,s,{startdate,enddate},[],field)
d = timeseries(c,s,{startdate,enddate},[],field,options,values)

d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval)
d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval,field)

d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime},
interval)
d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime},
interval,field)
```

Description

`d = timeseries(c,s,date)` retrieves raw tick data using the connection object and a security for a specific date.

`d = timeseries(c,s,date,interval,field)` retrieves raw tick data that is aggregated into intervals for a specific field.

`d = timeseries(c,s,date,[],field,options,values)` retrieves raw tick data without an aggregation interval for a specific field with the specified options and corresponding values.

`d = timeseries(c,s,{startdate,enddate})` retrieves raw tick data for a date range using a start date and an end date.

`d = timeseries(c,s,{startdate,enddate},interval,field)` retrieves raw tick data for a specific date range aggregated into intervals for a specific field.

`d = timeseries(c,s,{startdate,enddate},[],field)` retrieves raw tick data for a specific date range without an aggregation interval for a specific field.

`d = timeseries(c,s,{startdate,enddate},[],field,options,values)` retrieves raw tick data for a specific date range without an aggregation interval for a specific field with specified options and corresponding values.

`d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval)` retrieves raw trade tick data for a specific time range for each day within a specific date range, aggregated into intervals.

`d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval,field)` uses a specific field for tick data to return.

`d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime}, interval)` retrieves raw trade tick data for a whole day increment within a specific date and time range, aggregated into intervals.

`d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime}, interval,field)` uses a specific field for tick data to return.

Examples

Retrieve Tick Data for Specific Date and Pricing Source

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date. Use a security with and without a pricing source to retrieve tick data.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the trade tick series using the IBM security for today.

```
d = timeseries(c,'IBM US Equity',floor(now))
```

```
d =

    'TRADE'    [735537.40]    [181.69]    [100.00]
    'TRADE'    [735537.40]    [181.69]    [100.00]
    'TRADE'    [735537.40]    [181.68]    [100.00]
    ...
```

The columns in `d` are:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size

Here, the first row shows that 100 IBM shares sold for $181.69 today.

Retrieve the trade tick series using the Microsoft security with pricing source ETPX for today.

```
d = timeseries(c,'MSFT@ETPX US Equity',floor(now))
```

```
d =

    'TRADE'    [735537.40]    [35.53]    [100.00]
    'TRADE'    [735537.40]    [35.55]    [200.00]
    'TRADE'    [735537.40]    [35.55]    [100.00]
    ...
```

Here, the first row shows that 100 Microsoft shares are sold for $35.53 today.

Close the Bloomberg connection.

```
close(c)
```

Time Interval with Specific Field

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date. Specify the tick data to return using a time interval and field.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the trade tick series using the IBM security aggregated into 5-minute intervals for today.

```
d = timeseries(c,'IBM US Equity',floor(now),5,'Trade')
```

```
d =

  Columns 1 through 7

   735537.40        181.69        181.99        180.10        181.84     252322.00        861.00
   735537.40        181.90        181.97        181.57        181.65      78570.00        535.00
   735537.40        181.73        182.18        181.58        182.07     124898.00        817.00
     ...

  Column 8

   45815588.00
   14282076.00
   22710954.00
   ...
```

The columns in `d` contain the following:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

Here, the first row of data shows prices and tick data for the current date. The next row shows tick data for 5 minutes later.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Tick Data Using Option and Value

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date and field. Use option and value to return additional data.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the trade tick series using the `'F US Equity'` security without specifying the aggregation parameter for today. Also, return the condition codes.

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               'includeConditionCodes','true')
```

```
d =

    'TRADE'    [735556.57]    [17.12]    [   100.00]    'R6,IS'
    'TRADE'    [735556.57]    [17.12]    [   100.00]    ''
    'TRADE'    [735556.57]    [17.12]    [   500.00]    ''
    ...
```

The columns in `d` contain the following:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size
- Condition codes

Here, the first row shows that 100 `'F US Equity'` security shares sold for $17.12 today.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Tick Data Using Date Range

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the tick series for the `'F US Equity'` security for the last business day from the beginning of the day to noon.

```
d = timeseries(c,'F US Equity',{floor(now-4),floor(now-3.5)})
```

```
d =

    'TRADE'    [735552.67]    [17.09]    [   200.00]
    'TRADE'    [735552.67]    [17.09]    [   100.00]
    'TRADE'    [735552.67]    [17.09]    [   100.00]
   ...
```

The columns in `d` are:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size

Here, the first row shows that 200 `'F US Equity'` security shares were sold for $17.09 on the last business day.

Close the Bloomberg connection.

```
close(c)
```

Date Range with Interval and Specific Field

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range. Specify the interval and field.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Retrieve the trade tick series for the past 50 days for the IBM security aggregated into 5-minute intervals.

```
d = timeseries(c,'IBM US Equity',{floor(now)-50,floor(now)},5,'Trade')
```

```
ans =

  Columns 1 through 7

   735487.40        187.20        187.60        187.02        187.08     207683.00        560.00
   735487.40        187.03        187.13        186.65        186.78      46990.00        349.00
   735487.40        186.78        186.78        186.40        186.47      51589.00        399.00
     ...

Column 8

   38902968.00
    8779374.00
    9626896.00
   ...
```

The columns in `d` contain the following:

- Numeric representation of date and time
- Open price

- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row of data shows prices and tick data for the current date. The next row shows tick data for 5 minutes later.

Close the Bloomberg connection.

```
close(c)
```

Date Range with Numerous Fields

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range and numerous fields.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Return the Bid, Ask, and trade tick series for the security `'F US Equity'` for yesterday with a time interval at noon, without specifying the aggregation parameter.

```
d = timeseries(c,'F US Equity',{floor(now-1)+.5,floor(now-1)+.51},...
               [],{'Bid','Ask','Trade'})
```

```
d =

    'TRADE'    [735550.50]    [16.71]    [100.00]
    'ASK'      [735550.50]    [16.71]    [312.00]
    'BID'      [735550.50]    [16.70]    [177.00]
    ...
```

The columns in `d` are:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size

Here, the first row shows that 100 `'F US Equity'` security shares sold for $16.71 yesterday.

Close the Bloomberg connection.

```
close(c)
```

Date Range with Options and Values

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range. Specify options and values to return additional data.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE using `bpipe`.

Return the trade tick series for the security `'F US Equity'` for yesterday with a time interval at noon, without specifying the aggregation parameter. Also, return the condition codes, exchange codes, and broker codes.

```
d = timeseries(c,'F US Equity',{floor(now-1)+.5,floor(now-1)+.51},...
               [],'Trade',{'includeConditionCodes',...
               'includeExchangeCodes','includeBrokerCodes'},...
               {'true','true','true'})
```

```
d =

    'TRADE'    [735550.50]    [16.71]    [100.00]    'T'     'D'
    'TRADE'    [735550.50]    [16.70]    [400.00]    'IS'    'B'
    'TRADE'    [735550.50]    [16.70]    [100.00]    'IS'    'B'
    ...
```

The columns in `d` contain the following:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size
- Exchange condition codes
- Exchange codes

Broker codes are available for Canadian, Finnish, Mexican, Philippine, and Swedish equities only. In this case, the broker buy code appears in the seventh column and the broker sell code appears in the eighth column.

Here, the first row shows that 100 `'F US Equity'` security shares sold for $16.71 yesterday.

Close the Bloomberg connection.

```
close(c)
```

Date and Time Range with Interval

Use Bloomberg® to retrieve raw trade tick data by specifying a time range for each day in a specific date range. Specify the time interval for the tick data.

Create the Bloomberg® connection.

```
c = blp;
```

Alternatively, you can connect to Bloomberg® Server using `blpsrv` or Bloomberg® B-PIPE® using `bpipe`.

Retrieve the trade tick series for the `'F US Equity'` security for the last two days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. `d` is a numeric matrix.

```
s = 'F US Equity';
startdate = datetime('today')-1;
enddate = datetime('today');
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;

d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    736959.40         11.71         11.81         11.71         11.79
    736959.40         11.79         11.81         11.75         11.79
    736959.40         11.80         11.82         11.78         11.80

  Columns 6 through 8

   1375547.00       1190.00   16196757.00
    598924.00        898.00    7058724.00
    488655.00        641.00    5768371.50
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

Determine the maximum high price for the last two days.

```
highprices = d(:,3);
m = max(highprices)
```

```
m =

         11.82
```

Close the Bloomberg® connection.

```
close(c)
```

Date and Time Range with Interval and Specific Field

Use Bloomberg® to retrieve raw tick data by specifying a time range for each day in a specific date range. Specify the time interval and the field for the type of tick data to return. Here, specify the bid tick data.

Create the Bloomberg® connection.

```
c = blp;
```

Alternatively, you can connect to Bloomberg® Server using `blpsrv` or Bloomberg® B-PIPE® using `bpipe`.

Retrieve the tick series for the `'F US Equity'` security for the last two days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. Specify retrieving the bid tick series. `d` is a numeric matrix.

```
s = 'F US Equity';
startdate = datetime('today')-1;
enddate = datetime('today');
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;
field = 'BID';

d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval,field);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    736959.40         11.70         11.80         11.70         11.79
```

```
    736959.40         11.79         11.80         11.75         11.79
    736959.40         11.79         11.81         11.78         11.80

  Columns 6 through 8

    397711.00       1442.00    4681704.50
    450997.00       1698.00    5311330.50
    464761.00       1391.00    5481707.50
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

Determine the maximum high price for the last two days.

```
highprices = d(:,3);
m = max(highprices)
```

```
m =

         11.81
```

Close the Bloomberg® connection.

```
close(c)
```

Date and Time Range with Day Increment and Interval

Use Bloomberg® to retrieve raw trade tick data by specifying a time range for each day in a specific date range. Specify a day increment for the date range and the time interval for the tick data.

Create the Bloomberg® connection.

```
c = blp;
```

Alternatively, you can connect to Bloomberg® Server using `blpsrv` or Bloomberg® B-PIPE® using `bpipe`.

Retrieve the trade tick series for the `'IBM US Equity'` security for the last two months. Set the day increment to 5 days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. `d` is a numeric matrix.

```
s = 'IBM US Equity';
startdate = datetime('today')-60;
enddate = datetime('today');
dayincrement = 5;
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;

d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime}, ...
    interval);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    736900.40        147.00        147.04        146.55        146.62
    736900.40        146.62        146.87        146.62        146.71
    736900.40        146.72        146.79        146.52        146.54

  Columns 6 through 8

    125558.00        393.00   18440146.00
     39535.00        258.00    5800969.00
     49659.00        314.00    7282961.00
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

After the tick data for the first day in the date range, `d` contains tick data for a trading day that is 5 days later.

Close the Bloomberg® connection.

```
close(c)
```

Date and Time Range with Day Increment, Interval, and Specific Field

Use Bloomberg® to retrieve raw tick data by specifying a time range for each day in a specific date range. Specify a day increment for the date range, the time interval for the tick data, and the field for the type of tick data to return. Here, specify the bid tick data.

Create the Bloomberg® connection.

```
c = blp;
```

Alternatively, you can connect to Bloomberg® Server using `blpsrv` or Bloomberg® B-PIPE® using `bpipe`.

Retrieve the trade tick series for the `'F US Equity'` security for the last two months. Set the day increment to 5 days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. Specify the bid tick series. `d` is a numeric matrix.

```
s = 'F US Equity';
startdate = datetime('today')-60;
enddate = datetime('today');
dayincrement = 5;
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;
field = 'BID';

d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime}, ...
    interval,field);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    736900.40         11.50         11.54         11.49         11.50
    736900.40         11.50         11.50         11.48         11.48
    736900.40         11.48         11.49         11.44         11.44

  Columns 6 through 8

    422305.00       1158.00    4863894.00
    575966.00       1180.00    6617854.00
    288147.00       1489.00    3305491.75
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

After the tick data for the first day in the date range, `d` contains tick data for a trading day that is 5 days later.

Close the Bloomberg® connection.

```
close(c)
```

Return Tick Data as Table with Dates

Create a Bloomberg® connection, and then return intraday tick data. The `timeseries` function returns data for dates as a `datetime` array.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE® using `bpipe`.

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `timeseries` function returns data as a numeric array.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve the trade tick series for the IBM® security aggregated into 5-minute intervals for today. `d` is a table that contains the tick series data.

```
s = 'IBM US Equity';
date = floor(now);
interval = 5;
```

```
field = 'Trade';

d = timeseries(c,s,date,interval,field);
```

Access the first three ticks of data.

```
d(1:3,:)
```

```
ans =

  3×8 table

      DATE         OPEN      HIGH      LOW      CLOSE      VOLUME      NUMBER_OF_TICKS    TOTAL_
    ___________    ______    ______    ______    ______    _________    _______________    ______

    21-Dec-2017    153.17    153.31    153.08    153.31    152524.00        442.00         233676
    21-Dec-2017    153.35    153.35    152.82    152.84     46051.00        291.00          70486
    21-Dec-2017    152.84    153.21    152.82    153.16     30966.00        225.00          47373
```

`d` contains columns with the following data:

- Date
- Open price
- High price
- Low price
- Closing price
- Volume
- Number of ticks
- Total tick value in the bar

Access the first three dates in the `DATE` column.

```
d.DATE(1:3)
```

```
ans =

  3×1 datetime array

   21-Dec-2017
   21-Dec-2017
   21-Dec-2017
```

Close the Bloomberg connection.

```
close(c)
```

Return Tick Data as Timetable

Create a Bloomberg® connection, and then return intraday tick data. The `timeseries` function returns data for dates as a `timetable`.

Create the Bloomberg connection.

```
c = blp;
```

Alternatively, you can connect to the Bloomberg Server using `blpsrv` or Bloomberg B-PIPE® using `bpipe`.

Return data as a `timetable` by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `timeseries` function returns data as a numeric array.

```
c.DataReturnFormat = 'timetable';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve the trade tick series for the IBM® security aggregated into 5-minute intervals for today. `d` is a `timetable` that contains the tick series data.

```
s = 'IBM US Equity';
date = floor(now);
interval = 5;
field = 'Trade';

d = timeseries(c,s,date,interval,field);
```

Access the first three ticks of data.

```
d(1:3,:)
```

```
ans =

  3×7 timetable

      DATE         OPEN      HIGH      LOW      CLOSE      VOLUME      NUMBER_OF_TICKS    TOTAL_
    ___________    ______    ______    ______    ______    _________    _______________    ______

    21-Dec-2017    153.17    153.31    153.08    153.31    152524.00        442.00         23367
    21-Dec-2017    153.35    153.35    152.82    152.84     46051.00        291.00          7048
    21-Dec-2017    152.84    153.21    152.82    153.16     30966.00        225.00          4737
```

`d` is a `timetable` that contains the following data:

- Date
- Open price
- High price
- Low price
- Closing price
- Volume
- Number of ticks
- Total tick value in the bar

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg connection

connection object

Bloomberg connection, specified as a connection object created using `blp`, `blpsrv`, or `bpipe`.

`s` — Security

character vector | string scalar

Security, specified as a character vector or string scalar for a single Bloomberg security.

Data Types: `char` | `string`

`date` — Date

numeric scalar | character vector | string scalar | `datetime` array

Date, specified as a numeric scalar, character vector, string scalar, or `datetime` array. `date` specifies the date for the returned tick data based on the entire day from midnight until 11:59:59 p.m.

Example: `floor(now)`

Data Types: `double` | `char` | `string` | `datetime`

`interval` — Time interval

numeric scalar

Time interval, specified as a numeric scalar to denote the number of minutes between ticks for the returned tick data.

Data Types: `double`

`field` — Bloomberg field

`'TRADE'` (default) | `'BID'` | `'ASK'` | ...

Bloomberg field, specified as one of these values that define the tick data to return.

Request Type	Valid Bloomberg Field Values
IntradayBarRequest with time interval specified	`'TRADE'`
	`'BID'`
	`'ASK'`
	`'BID_BEST'`
	`'ASK_BEST'`
IntradayTickRequest with no time interval specified	`'TRADE'`
	`'BID'`
	`'ASK'`
	`'BID_BEST'`
	`'ASK_BEST'`
	`'SETTLE'`

`options` — Bloomberg API options

`'includeConditionCodes'` | `'includeExchangeCodes'` | `'includeBrokerCodes'` | ...

Bloomberg API options, specified as one of the values in this table.

Value	Description
`'includeConditionCodes'`	Exchange condition codes associated with the event
`'includeExchangeCodes'`	Exchange code where tick originated
`'includeBrokerCodes'`	Broker code
`'includeRpsCodes'`	Reporting party side
`'includeNonPlottableEvents'`	After-hours data

Note The value `'includeNonPlottableEvents'` applies to raw intraday requests only.

To specify more than one Bloomberg API option, use a cell array of these values.

Specify the corresponding Bloomberg API value for each API option. The number of options must match the number of values.

For example, to specify one Bloomberg API option, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               'includeConditionCodes','true');
```

To specify two Bloomberg API options, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               {'includeConditionCodes','includeExchangeCodes'},...
               {'true','true'});
```

For details about the options, see the *Bloomberg API Developer's Guide*.

Data Types: `char` | `cell`

`values` — Bloomberg API values

`'true'` | `'false'`

Bloomberg API values, specified as `'true'` or `'false'`. Each value corresponds to the specified Bloomberg API option. To specify more than one Bloomberg API value, use a cell array. The number of values must match the number of options.

For example, to specify one Bloomberg API option, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               'includeConditionCodes','true');
```

To specify two Bloomberg API options, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               {'includeConditionCodes','includeExchangeCodes'},...
               {'true','true'});
```

Data Types: `char` | `cell`

`startdate` — Start date
numeric scalar | character vector | string scalar | `datetime` array

Start date, specified as a numeric scalar, character vector, string scalar, or `datetime` array. This date specifies the beginning of the date range for the returned tick data. If no ticks are present in the date range, then returned tick data is empty.

Example: `floor(now-1)`

Data Types: `double` | `char` | `string` | `datetime`

`enddate` — End date
numeric scalar | character vector | string scalar | `datetime` array

End date, specified as a numeric scalar, character vector, string scalar, or `datetime` array. This date specifies the end of the date range for the returned tick data. If no ticks are present in the date range, then returned tick data is empty.

Example: `floor(now)`

Data Types: `double` | `char` | `string` | `datetime`

`starttime` — Start time
character vector | string scalar | `datetime` array

Start time, specified as a character vector, string scalar, or `datetime` array. This time specifies the start time of the time range for the returned tick data.

Example: `'09:30:00'`

Data Types: `char` | `string` | `datetime`

`endtime` — End time
character vector | string scalar | `datetime` array

End time, specified as a character vector, string scalar, or `datetime` array. This time specifies the end time of the time range for the returned tick data.

Example: `'16:30:00'`

Data Types: `char` | `string` | `datetime`

`dayincrement` — Day increment
1 (default) | numeric scalar

Day increment, specified as a numeric scalar. This number specifies the whole day increment for a specific date range. For example, if the day increment is 7, then the returned data contains ticks for every 7th day starting from the first day within the date range.

Data Types: `double`

Output Arguments

`d` — Bloomberg tick data
cell array | numeric array | table | timetable

Bloomberg tick data, returned as one of these data types:

- Cell array for requests without a specified time interval (raw tick data)
- Numeric array for requests with a specified time interval
- table
- timetable

The data type of the tick data depends on the DataReturnFormat and DatetimeType properties of the connection object.

Note The Bloomberg API returns the tick time with precision in seconds.

Limitations

When the data request is too large, `timeseries` displays this error message:

```
Timeout error:
Error using blp/timeseries>processResponseEvent (line 338) REQUEST FAILED: responseError = {

source = bbdbl7

code = -2

category = TIMEOUT

message = Timed out getting data from store [nid:327]

subcategory = INTERNAL_ERROR

}
```

To fix this error, shorten the length of the date range by modifying the input arguments `startdate` and `enddate`.

Tips

- For better performance, add the Bloomberg file `blpapi3.jar` to the MATLAB static Java class path by modifying the file `$MATLAB/toolbox/local/javaclasspath.txt`. For details about the static Java class path, see “Static Path of Java Class Path”.
- You cannot retrieve Bloomberg intraday tick data for a date more than 140 days ago.
- The *Bloomberg API Developer’s Guide* states that `'TRADE'` corresponds to LAST_PRICE for IntradayTickRequest and IntradayBarRequest.
- Bloomberg V3 intraday tick data supports additional name-value pairs. For details on these pairs, see the *Bloomberg API Developer’s Guide* by typing `WAPI` and clicking the **<GO>** button on the Bloomberg terminal.
- You can check data and field availability by using the Bloomberg Excel Add-In.

Version History

Introduced in R2010a

See Also

`blp` | `getdata` | `history` | `realtime` | `close`

Topics

“Retrieve Bloomberg Intraday Tick Data” on page 3-11
“Workflow for Bloomberg” on page 3-15

bloomberg

Bloomberg Desktop connection V3

Description

The `bloomberg` function creates a `bloomberg` object. The `bloomberg` object represents a Bloomberg Desktop connection using the Bloomberg V3 C++ API.

Other Datafeed Toolbox functions connect to different Bloomberg services: Bloomberg Server (`bloombergServer`) and Bloomberg B-PIPE (`bloombergBPIPE`). For details about these services, see “Comparing Bloomberg Connections” on page 2-4.

For details about Bloomberg connection requirements, see “Data Server Connection Requirements” on page 1-3. To ensure a successful Bloomberg connection, perform the required steps before executing `bloomberg`. For details, see “Installing Bloomberg and Configuring Connections” on page 1-5.

Creation

Syntax

```
c = bloomberg
c = bloomberg(port,ip,timeout)
```

Description

`c = bloomberg` creates a Bloomberg connection object with the Bloomberg Desktop C++ interface. You need a Bloomberg Desktop software license for the machine running the Datafeed Toolbox and MATLAB.

`c = bloomberg(port,ip,timeout)` sets the port and timeout properties, and uses the IP address of the local machine running Bloomberg to create a Bloomberg connection.

Caution To refer to a Bloomberg connection in other functions, use the connection object created by the `bloomberg` function. Otherwise, using `bloomberg` as an input argument opens multiple Bloomberg connections, causing unexpected behavior and exhausting memory resources.

Input Arguments

`ip` — IP address
`[ ]` (default) | character vector | string scalar

IP address that identifies the local machine running Bloomberg, specified as a character vector or string scalar.

Example: `'localhost'`

Data Types: `char` | `string`

Properties

`Session` — Bloomberg V3 session

Bloomberg V3 API Session object

This property is read-only.

Bloomberg V3 session, specified as a Bloomberg V3 API Session object.

Example: `[1x1 datafeed.internal.BLPSession]`

`Port` — Port number of local machine

`[]` (default) | numeric scalar

Port number of the local machine running Bloomberg, specified as a numeric scalar.

Example: `8194`

Data Types: `double`

`IPAddress` — IP address of local machine

`[]` (default) | character vector

IP address of the local machine running Bloomberg, specified as a character vector.

The `bloomberg` function sets this property using the `ip` input argument.

Example: `'localhost'`

Data Types: `char`

`TimeOut` — Timeout

numeric scalar

Timeout specifying the time in milliseconds that MATLAB attempts to connect to Bloomberg Desktop before timing out, specified as a numeric scalar.

Example: `10000`

Data Types: `double`

`DatetimeType` — Date and time data type

`''` (default) | `'datetime'`

Date and time data type, specified as one of these values.

Value	Description
`''` (default)	Return date and time values as MATLAB date numbers.
`'datetime'`	Return date and time values as a `datetime` array.

You can specify these values using a character vector or string (for example, `"datetime"`).

When you create a `bloomberg` object, the `bloomberg` function leaves this property unset. To retrieve data, you must set this property value manually at the command line or in a script using dot notation, for example:

```
c.DatetimeType = 'datetime';
```

Then, you can use these supported functions:

- getbulkdata
- getdata
- history
- tahistory
- timeseries

> **Note** If the `DataReturnFormat` property value is `'table'` and the `DatetimeType` property value is `'datetime'`, then the returned data is a table that contains date and time values as a `datetime` array. If the `DataReturnFormat` property value is an empty character vector, then setting the `DatetimeType` property to `'datetime'` returns date and time values for aggregated ticks and historical requests as MATLAB date numbers.

`DataReturnFormat` — Data return format
`'cell'` | `'structure'` | `'table'` | `'timetable'`

Data return format, specified as one of these values, which determine the data type of the returned data.

Value	Data Type of Returned Data
`'cell'`	`cell array`
`'table'`	`table`
`'timetable'`	`timetable`
`'structure'`	`structure`

Note The default data type of the returned data depends on the executed function. To specify the default data type, set the `DataReturnFormat` property to `''`. For default data types, see the supported function list.

You can specify these values using a character vector or string (for example, `"table"`).

When you create a `bloomberg` object, the `bloomberg` function leaves this property unset. To retrieve data, you must set this property value manually at the command line or in a script using dot notation, for example:

```
c.DataReturnFormat = 'structure';
```

Then, you can use these supported functions.

Supported Function	Valid Data Types for Returned Data
category	• `cell array` (default) • `structure` • `table`
eqs	• `cell array` (default) • `structure` • `table`
fieldinfo	• `cell array` (default) • `structure` • `table`
fieldsearch	• `cell array` (default) • `structure` • `table`
lookup	• `structure` (default) • `table`
portfolio	• `structure` (default) • `table`
getbulkdata	• `structure` (default) • `table` • `timetable`
getdata	• `structure` (default) • `table` • `timetable`
history	• numeric array (default) • `table` • `timetable`
tahistory	• `structure` (default) • `table` • `timetable`
timeseries	• `cell array` (default for raw tick data) • numeric array (default for interval tick data) • `table` • `timetable`

Note Regardless of the `DatetimeType` property value, if the `DataReturnFormat` property value is `'timetable'`, then the `getdata` and `getbulkdata` functions return a table that contains date and time values as `datetime` arrays.

Object Functions

Bloomberg Desktop Connection

close	Close Bloomberg Desktop connection V3
isconnection	Determine Bloomberg Desktop connection V3

Bloomberg Desktop Data Retrieval

eqs	Equity screening data for Bloomberg Desktop connection V3
get	Properties of Bloomberg Desktop connection V3
getbulkdata	Bulk data with header information for Bloomberg Desktop connection V3
getdata	Current data for Bloomberg Desktop connection V3
history	Historical data for Bloomberg Desktop connection V3
portfolio	Current portfolio data for Bloomberg Desktop connection V3
realtime	Real-time data for Bloomberg Desktop connection V3
tahistory	Historical technical analysis for Bloomberg Desktop connection V3
timeseries	Intraday tick data for Bloomberg Desktop connection V3

Retrieve Bloomberg Desktop Information

category	Field category search for Bloomberg Desktop connection V3
fieldinfo	Field information for Bloomberg Desktop connection V3
fieldsearch	Field search for Bloomberg Desktop connection V3
lookup	Find information about securities for Bloomberg Desktop connection V3

Examples

Create Bloomberg Desktop Connection

First, create a Bloomberg Desktop connection. Then, request last and open prices for a security. The current data you see when running this code can differ from the output data here.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg
```

```
c =

  bloomberg with properties:

    Session: [1×1 datafeed.internal.BLPSession]
    IPAddress: "localhost"
    Port: 8194.00
    TimeOut: 0
    DatetimeType: ''
    DataReturnFormat: ''
```

The `bloomberg` function creates a `bloomberg` object `c` with these properties:

- Bloomberg V3 API Session object
- IP address of the local machine
- Port number of the local machine
- Number of milliseconds specifying how long MATLAB® attempts to connect to Bloomberg Desktop before timing out
- Date and time data type
- Data return format

Request last and open prices for Microsoft.

```
[d,sec] = getdata(c,'MSFT US Equity',{'LAST_PRICE';'OPEN'})
```

```
d =
    LAST_PRICE: 33.3401
          OPEN: 33.6000

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the name of the security in `sec`.

Close the Bloomberg connection.

```
close(c)
```

Create Bloomberg Desktop Connection with Timeout

First, create a Bloomberg Desktop connection with a timeout value. Then, request last and open prices for a security. The current data you see when running this code can differ from the output data here.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface. Specify a timeout value of 10,000 milliseconds.

```
c = bloomberg([],[],10000)
```

```
c =

  bloomberg with properties:

    Session: [1×1 datafeed.internal.BLPSession]
    IPAddress: "localhost"
    Port: 8194.00
    TimeOut: 10000
    DatetimeType: ''
    DataReturnFormat: ''
```

The `bloomberg` function creates a `bloomberg` object `c` with these properties:

- Bloomberg V3 API Session object
- IP address of the local machine

- Port number of the local machine
- Number of milliseconds specifying how long MATLAB® attempts to connect to Bloomberg Desktop before timing out
- Date and time data type
- Data return format

Request last and open prices for Microsoft.

```
[d,sec] = getdata(c,'MSFT US Equity',{'LAST_PRICE';'OPEN'})
```

```
d =
    LAST_PRICE: 33.3401
          OPEN: 33.6000

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the name of the security in `sec`.

Close the Bloomberg connection.

```
close(c)
```

Version History

Introduced in R2021a

See Also

Topics

“Data Server Connection Requirements” on page 1-3
“Comparing Bloomberg Connections” on page 2-4
“Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15

category

Field category search for Bloomberg Desktop connection V3

Syntax

```
d = category(c,f)
```

Description

`d = category(c,f)` returns category information given the search term `f` using the Bloomberg Desktop C++ interface.

Examples

Search for Bloomberg Last Price Field

Create a Bloomberg connection, and then request the category description of the last price field.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return data as a table by setting the `DataReturnFormat` property of the `bloomberg` object. If you do not set this property, the `category` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Request the Bloomberg category description of the last price field.

```
f = 'LAST_PRICE';
d = category(c,f);
```

Display the first three rows of the Bloomberg category description data in `d`.

```
d(1:3,:)
```

```
ans =

  3×5 table

    CATEGORY       ID                  MNEMONIC                          DESCRIPTION
    __________    _______    ____________________________    ____________________________________

    'Analysis'    'OP179'    'THETA_LAST'                    'Theta Last Price'
    'Analysis'    'VM048'    'DDMX_PERCENT_CHANGE_LAST_PRICE'    'DDMX Percent Change Last Price
    'Analysis'    'YL005'    'YLD_CNV_LAST'                  'Last Yield To Convention'
```

The columns in `d` are:

- Category
- Field identifier
- Field mnemonic
- Field name
- Field data type

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

f — Search term

character vector | string scalar

Search term, specified as a character vector or string scalar to denote Bloomberg fields.

Data Types: `char` | `string`

Output Arguments

d — Category data

cell array (default) | structure | table

Category data, returned as an N-by-5 cell array, a structure, or a table.

The columns (or fields) of the data types are:

- Category
- Field identifier
- Field mnemonic
- Field name
- Field data type

The data type of the category data depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `getdata` | `history` | `realtime` | `timeseries` | `fieldinfo` | `fieldsearch`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface” on page 5-18
“Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15

close

Close Bloomberg Desktop connection V3

Syntax

```
close(c)
```

Description

`close(c)` closes the Bloomberg connection V3 `c` using the Bloomberg Desktop C++ interface.

Examples

Close Bloomberg Connection

First, create a Bloomberg Desktop connection. Then, request last and open prices for a security and close the connection. The current data you see when running this code can differ from the output data here.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Request last and open prices for Microsoft.

```
[d,sec] = getdata(c,'MSFT US Equity',{'LAST_PRICE';'OPEN'})
```

```
d =
    LAST_PRICE: 33.3401
          OPEN: 33.6000

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the security in `sec`.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `isconnection` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface” on page 5-18
“Retrieve Bloomberg Historical Data Using Bloomberg Desktop C++ Interface” on page 5-20
“Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15
“Retrieve Bloomberg Intraday Tick Data Using Bloomberg Desktop C++ Interface” on page 5-24
“Retrieve Bloomberg Real-Time Data Using Bloomberg Desktop C++ Interface” on page 5-26

eqs

Equity screening data for Bloomberg Desktop connection V3

Syntax

```
d = eqs(c,sname)
d = eqs(c,sname,stype)
d = eqs(c,sname,stype,languageid)
d = eqs(c,sname,stype,languageid,group)
d = eqs(c,sname,stype,languageid,group,'OverrideFields',ov)
```

Description

`d = eqs(c,sname)` returns equity screening data given the Bloomberg V3 session screen name `sname` using the Bloomberg Desktop C++ interface.

`d = eqs(c,sname,stype)` also specifies the screen type `stype`.

`d = eqs(c,sname,stype,languageid)` also specifies the language identifier `languageid`.

`d = eqs(c,sname,stype,languageid,group)` also specifies the optional group identifier `group`.

`d = eqs(c,sname,stype,languageid,group,'OverrideFields',ov)` also specifies the Bloomberg override fields and values `ov`.

Examples

Retrieve Equity Screening Data for Screen

Create a Bloomberg connection, and then retrieve frontier market stock data.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return data as a table by setting the `DataReturnFormat` property of the `bloomberg` object. If you do not set this property, the `eqs` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Retrieve equity screening data for the screen named `Frontier Market Stocks with 1 billion USD Market Caps`.

```
sname = 'Frontier Market Stocks with 1 billion USD Market Caps';
d = eqs(c,sname);
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =
```

```
3×8 table

     Cntry                 Name                  IndGroup    MarketCap     Price_D_1     P_B
    ___________    ___________________________    ________    __________    _________    _____

    'Venezuela'    'MERCANTIL SERVICIOS FINAN-A'    'Banks'     7.3424e+12      70088      278.25
    'Venezuela'    'BANCO DEL CARIBE-A'             'Banks'     2.0442e+12      24531      2321.8
    'Venezuela'    'BANCO PROVINCIAL'               'Banks'     1.2632e+12      11715      52.344
```

The columns in d are:

- Country name
- Company name
- Industry name
- Market capitalization
- Price
- Price-to-book ratio
- Price-earnings ratio
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Retrieve Equity Screening Data for a Screen Type

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve equity screening data for the screen called `Vehicle-Engine-Parts` and the screen type equal to `'GLOBAL'`.

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL');
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    'Ticker'        'Short Name'        'Market Cap'        'Price:D-1'    'P/E'
    'HON     US'    'HONEYWELL INTL'    [69451382784.00]    [    88.51]    [16.81]
    'CMI     US'    'CUMMINS INC'       [24799526912.00]    [   132.36]    [17.28]

  Columns 6 through 8

    'Total Return YTD'    'Revenue T12M'      'EPS T12M'
    [           42.43]    [38248998912.00]    [    4.11]
    [           24.43]    [17004999936.00]    [    7.57]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen. The first row contains column headers. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Retrieve Equity Screening Data for a Screen in German

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve equity screening data for the screen called `Vehicle-Engine-Parts`, the screen type equal to `'GLOBAL'`, and return data in German.

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL','GERMAN');
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
  Columns 1 through 5

    'Ticker'        'Kurzname'          'Marktkapitalisie...'    'Preis:D-1'    'KGV'
    'HON     US'    'HONEYWELL INTL'    [     69451382784.00]    [    88.51]    [16.81]
    'CMI     US'    'CUMMINS INC'       [     24799526912.00]    [   132.36]    [17.28]

  Columns 6 through 8

    'Gesamtertrag YTD'    'Erlös T12M'        'EPS T12M'
    [           42.43]    [38248998912.00]    [    4.11]
    [           24.43]    [17004999936.00]    [    7.57]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen. The first row contains column headers in German. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Retrieve Equity Screening Data for a Screen with a Specified Screen Folder Name

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve equity screening data for the Bloomberg screen called `Vehicle-Engine-Parts`, using the Bloomberg screen type `'GLOBAL'` and the language `'ENGLISH'`, and the Bloomberg screen folder name `'GENERAL'`.

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL','ENGLISH','GENERAL');
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    'Ticker'        'Short Name'        'Market Cap'        'Price:D-1'    'P/E'
    'HON     US'    'HONEYWELL INTL'    [69451382784.00]    [    88.51]    [16.81]
    'CMI     US'    'CUMMINS INC'       [24799526912.00]    [   132.36]    [17.28]

  Columns 6 through 8

    'Total Return YTD'    'Revenue T12M'      'EPS T12M'
    [           42.43]    [38248998912.00]    [    4.11]
    [           24.43]    [17004999936.00]    [    7.57]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen. The first row contains column headers. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Retrieve Equity Screening Data Using Override Fields

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve equity screening data as of a specified date using these input arguments. The override field `PiTDate` is equivalent to the flag `AsOf` in the Bloomberg Excel Add-In.

- Bloomberg connection `c`
- Bloomberg screen is `Vehicle-Engine-Parts`
- Bloomberg screen type is `'GLOBAL'`
- Language is `'ENGLISH'`
- Bloomberg screen folder name is `'GENERAL'`
- Override field `PiTDate` is September 9, 2014

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL','ENGLISH','GENERAL',...
    'OverrideFields',{'PiTDate','20140909'});
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    'Ticker'        'Short Name'        'Market Cap'    'Price:D-1'       'P/E'
    'HON US'     'HONEYWELL INTL'      [7.3919e+10]    [  94.4600]    [17.8087]
    'TSLA US'    'TESLA MOTORS'        [3.4707e+10]    [ 278.4800]    [    NaN]

  Columns 6 through 8

    'Total Return YTD'    'Revenue T12M'    'EPS T12M'
    [          4.8907]    [  3.9966e+10]    [  5.1600]
    [         85.1239]    [  2.4365e+09]    [ -1.3500]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen as of September 9, 2014. The first row contains column headers. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

`sname` — Screen name
character vector | string scalar

Screen name, specified as a character vector or string scalar to denote the Bloomberg V3 session screen name to execute. The screen can be a customized equity screen or one of the Bloomberg example screens accessed by using the **EQS <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

`stype` — Screen type
`'GLOBAL'` | `'PRIVATE'`

Screen type, specified as one of the two preceding values to denote the Bloomberg screen type. `'GLOBAL'` denotes a Bloomberg screen name and `'PRIVATE'` denotes a customized screen name. When using the optional `group` input argument, `stype` cannot be set to `'PRIVATE'` for customized screen names.

`languageid` — Language identifier
character vector | string scalar

Language identifier, specified as a character vector or string to denote the language for the returned data. This argument is optional.

Data Types: `char` | `string`

`group` — Group identifier
character vector | string scalar

Group identifier, specified as a character vector or string to denote the Bloomberg screen folder name accessed by using the **EQS <GO>** option from the Bloomberg terminal. This argument is optional. When using this argument, `stype` cannot be set to `'PRIVATE'` for customized screen names.

Data Types: `char` | `string`

`ov` — Bloomberg override field values
cell array

Bloomberg override field values, specified as an `n`-by-`2` cell array. The first column of the cell array is the override field. The second column is the override value.

Example: `{'PiTDate','20140909'}`

Data Types: `cell`

Output Arguments

`d` — Equity screening data
cell array (default) | structure | table

Equity screening data, returned as a cell array, structure, or table. The data type of the equity screening data depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `getdata` | `tahistory`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface” on page 5-18
“Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15

fieldinfo

Field information for Bloomberg Desktop connection V3

Syntax

```
d = fieldinfo(c,f)
```

Description

`d = fieldinfo(c,f)` returns field information using the `bloomberg` object `c` with the Bloomberg Desktop C++ interface and field mnemonic `f`.

Examples

Retrieve Information for Last Price Field

Create a Bloomberg connection, and then retrieve information for the last price field.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return data as a table by setting the `DataReturnFormat` property of the `bloomberg` object. If you do not set this property, the `fieldinfo` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Retrieve the Bloomberg field information for the `LAST_PRICE` field.

```
f = 'LAST_PRICE';
d = fieldinfo(c,f);
```

Display the last four columns in the returned Bloomberg information.

```
d(:,2:5)
```

```
ans =

  1×4 table

      ID          MNEMONIC             DESCRIPTION           DATATYPE
    ________    _____________    ____________________________    ________

    'RQ005'     'LAST_PRICE'     'Last Trade/Last Price'     'Double'
```

The columns in `d` are:

- Field identifier

- Field mnemonic
- Field name
- Field data type

You can also access the Bloomberg help information in the first column.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

f — Field mnemonic

character vector | string scalar

Field mnemonic, specified as a character vector or string scalar that denotes the Bloomberg field information to retrieve.

Data Types: `char` | `string`

Output Arguments

d — Field information

cell array (default) | structure | table

Field information, returned as an N-by-5 cell array, a structure, or a table.

The columns (or fields) of the data types are:

- Field help
- Field identifier
- Field mnemonic
- Field name
- Field data type

The data type of the field information depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `getdata` | `history` | `realtime` | `timeseries` | `category` | `fieldsearch`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface” on page 5-18
“Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15

fieldsearch

Field search for Bloomberg Desktop connection V3

Syntax

```
d = fieldsearch(c,f)
```

Description

`d = fieldsearch(c,f)` returns field information using the `bloomberg` object `c` with the Bloomberg Desktop C++ interface and search term `f`.

Examples

Retrieve Information for Last Price Field

Create a Bloomberg connection, and then retrieve information for the last price field.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return data as a table by setting the `DataReturnFormat` property of the `bloomberg` object. If you do not set this property, the `fieldsearch` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Return information for the search term `LAST_PRICE`.

```
f = 'LAST_PRICE';
d = fieldsearch(c,f);
```

Display the first three rows of the field information in `d`.

```
d(1:3,:)
```

```
ans =

  3×5 table

          CATEGORY             ID           MNEMONIC                DESCRIPTION
    ___________________________    _______    ____________________    ___________________________________

    'Market Activity/Last'       'PR005'    'PX_LAST'               'Last Price'
    'Market Activity/Last'       'RQ005'    'LAST_PRICE'            'Last Trade/Last Price'
    'Market Activity/Last'       'PR910'    'CRNCY_ADJ_PX_LAST'     'Currency Adjusted Last Price'
```

The columns in `d` are:

- Category
- Field identifier
- Field mnemonic
- Field name
- Field data type

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

f — Search term

character vector | string scalar

Search term, specified as a character vector or string scalar that denotes the Bloomberg field descriptive data to retrieve.

Data Types: `char` | `string`

Output Arguments

d — Field data

cell array (default) | structure | table

Field data, returned as an N-by-5 cell array, a structure, or a table.

The columns (or fields) of the data types are:

- Category
- Field identifier
- Field mnemonic
- Field name
- Field data type

The data type of the field data depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `getdata` | `history` | `realtime` | `timeseries` | `category` | `fieldinfo`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface” on page 5-18

“Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15

get

Properties of Bloomberg Desktop connection V3

Syntax

```
v = get(c)
v = get(c,properties)
```

Description

`v = get(c)` returns a structure where each field name is the name of a property of the `bloomberg` object `c`, which uses the Bloomberg Desktop C++ interface, and each field contains the value of that property.

`v = get(c,properties)` returns the value of the specified properties `properties` for the `bloomberg` object.

Examples

Retrieve Bloomberg Connection Properties

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the Bloomberg connection properties.

```
v = get(c)
```

```
v =

  struct with fields:

      session: [1×1 datafeed.internal.BLPSession]
    ipaddress: "localhost"
         port: 8194.00
```

`v` is a structure containing the Bloomberg session object, IP address, port number, timeout value, date and time data type, and data return format.

Close the Bloomberg connection.

```
close(c)
```

Retrieve One Bloomberg Connection Property

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the port number from the Bloomberg connection object by specifying `'port'` as a character vector.

```
property = "port";
v = get(c,property)
```

```
v =

        8194
```

`v` is a double that contains the port number of the Bloomberg connection object.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Two Bloomberg Connection Properties

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Create a cell array `properties` with character vectors `'session'` and `'port'`. Retrieve the Bloomberg session object and port number from the Bloomberg connection object.

```
properties = ["session" "port"];
v = get(c,properties)
```

```
v =

  struct with fields:

    session: [1×1 datafeed.internal.BLPSession]
       port: 8194
```

`v` is a structure containing the Bloomberg session object and port number.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

`properties` — Property names

character vector | string scalar | cell array of character vectors | string array

Property names, specified as a character vector, string scalar, cell array of character vectors, or string array containing Bloomberg connection property names. The property names are `session`, `ipaddress`, `port`, and `timeout`.

Data Types: `char` | `cell` | `string`

Output Arguments

v — Bloomberg connection properties

numeric scalar | character vector | object | structure

Bloomberg connection properties, returned as these data types depending on the requested properties.

Requested Properties	Data Type
Port number or timeout	Numeric scalar
IP address	Character vector
Bloomberg session	Object
All properties	Structure

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15

getbulkdata

Bulk data with header information for Bloomberg Desktop connection V3

Syntax

```
d = getbulkdata(c,s,f)
d = getbulkdata(c,s,f,o,ov)
d = getbulkdata(c,s,f,o,ov,Name,Value)
[d,sec] = getbulkdata( ___ )
```

Description

`d = getbulkdata(c,s,f)` returns the bulk data for the fields `f` for the security list `s` using the `bloomberg` object `c` with the Bloomberg Desktop C++ interface.

`d = getbulkdata(c,s,f,o,ov)` returns the bulk data using the override fields `o` with corresponding override values `ov`.

`d = getbulkdata(c,s,f,o,ov,Name,Value)` returns the bulk data with additional options specified by one or more name-value pair arguments for Bloomberg request settings.

`[d,sec] = getbulkdata( ___ )` additionally returns the security list `sec` using any of the input argument combinations in the previous syntaxes.

Examples

Return Specific Field for Given Security

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return the dividend history for IBM.

```
security = 'IBM US Equity';
field = 'DVD_HIST'; % Dividend history field

[d,sec] = getbulkdata(c,security,field)
```

```
d =

    DVD_HIST: {{149x7 cell}}

sec =

    'IBM US Equity'
```

`d` is a structure with one field that contains a cell array with the returned bulk data. `sec` contains the IBM security name.

Display the dividend history with the associated header information by accessing the structure field `DVD_HIST`. This field is a cell array that contains one cell array. The nested cell array contains the dividend history data. Access the contents of the nested cell using cell array indexing.

```
d.DVD_HIST{1}
```

```
ans =

  Columns 1 through 6

    'Declared Date'    'Ex-Date'    'Record Date'    'Payable Date'    'Dividend Amount'    'Dividend Frequency'
    [       735536]    [ 735544]    [     735546]    [      735578]    [         0.95]      'Quarter'
    [       735445]    [ 735453]    [     735455]    [      735487]    [         0.95]      'Quarter'
    [       735354]    [ 735362]    [     735364]    [      735395]    [         0.95]      'Quarter'
    ...

  Column 7

    'Dividend Type'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
    ...
```

The first row of the dividend history data is the header information that describes the contents of each column.

Close the Bloomberg connection.

```
close(c)
```

Return Specific Field Using Override Values

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return the dividend history for IBM with dividend dates from January 1, 2004, through January 1, 2005.

```
security = 'IBM US Equity';
field = 'DVD_HIST';                          % Dividend history field
override = {'DVD_START_DT','DVD_END_DT'};  % Dividend start and
                                             % End dates
overridevalues = {'20040101','20050101'};

[d,sec] = getbulkdata(c,security,field,override,overridevalues)
```

```
d =

    DVD_HIST: {{5x7 cell}}

sec =

    'IBM US Equity'
```

`d` is a structure with one field that contains a cell array with the returned bulk data. `sec` contains the IBM security name.

Display the dividend history with the associated header information by accessing the structure field `DVD_HIST`. This field is a cell array that contains one cell array. The nested cell array contains the dividend history data. Access the contents of the nested cell using cell array indexing.

```
d.DVD_HIST{1}
```

```
ans =

  Columns 1 through 6

    'Declared Date'    'Ex-Date'    'Record Date'    'Payable Date'    'Dividend Amount'    'Dividend Frequency'
    [       732246]    [ 732259]    [     732261]    [      732291]    [         0.18]      'Quarter'
    [       732155]    [ 732165]    [     732169]    [      732200]    [         0.18]      'Quarter'
    [       732064]    [ 732073]    [     732077]    [      732108]    [         0.18]      'Quarter'
    [       731973]    [ 731983]    [     731987]    [      732016]    [         0.16]      'Quarter'

  Column 7

    'Dividend Type'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
```

The first row of the dividend history data is the header information that describes the contents of each column.

Close the Bloomberg connection.

```
close(c)
```

Return Specific Field Using Name-Value Pair Arguments

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return the closing price and dividend history for IBM with dividend dates from January 1, 2004, through January 1, 2005. Specify the data return format as a character vector by setting the name-value pair argument `'returnFormattedValue'` to `'true'`.

```
security = 'IBM US Equity';
fields = {'LAST_PRICE','DVD_HIST'};        % Closing price and
                                           % Dividend history fields
override = {'DVD_START_DT','DVD_END_DT'};  % Dividend start and
                                           % End dates
overridevalues = {'20040101','20050101'};

[d,sec] = getbulkdata(c,security,fields,override,overridevalues,...
                      'returnFormattedValue',true)
```

```
d =

      DVD_HIST: {{5x7 cell}}
    LAST_PRICE: {'188.74'}

sec =

    'IBM US Equity'
```

`d` is a structure with two fields. The first field `DVD_HIST` contains a cell array with the dividend historical data as a cell array. The second field `LAST_PRICE` contains a cell array with the closing price as a character vector. `sec` contains the IBM security name.

Display the closing price.

```
d.LAST_PRICE
```

```
ans =

    '188.74'
```

Display the dividend history with the associated header information by accessing the structure field `DVD_HIST`. This field is a cell array that contains one cell array. The nested cell array contains the dividend history data. Access the contents of the nested cell using cell array indexing.

```
d.DVD_HIST{1}
```

```
ans =

  Columns 1 through 6

    'Declared Date'    'Ex-Date'    'Record Date'    'Payable Date'    'Dividend Amount'    'Dividend Frequency'
    [       732246]    [ 732259]    [     732261]    [      732291]    [         0.18]      'Quarter'
    [       732155]    [ 732165]    [     732169]    [      732200]    [         0.18]      'Quarter'
    [       732064]    [ 732073]    [     732077]    [      732108]    [         0.18]      'Quarter'
    [       731973]    [ 731983]    [     731987]    [      732016]    [         0.16]      'Quarter'

  Column 7

    'Dividend Type'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
```

The first row of the dividend history data is the header information that describes the contents of each column.

Close the Bloomberg connection.

```
close(c)
```

Return Bulk Data as Table with Datetime

Create a Bloomberg connection, and then request dividend history data. The `getbulkdata` function returns data for dates as a `datetime` array.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `getbulkdata` function returns data as a structure.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Return the dividend history for IBM.

```
s = 'IBM US Equity';
f = 'DVD_HIST'; % Dividend history field

d = getbulkdata(c,s,f);
```

Display the first three rows of the table.

```
d.DVD_HIST{1}(1:3,:)
```

```
ans =

  3×7 table

       DeclaredDate            ExmDate            RecordDate           PayableDate
    ____________________    ____________________    ____________________    ____________________

    31-Oct-2017 00:00:00    09-Nov-2017 00:00:00    10-Nov-2017 00:00:00    09-Dec-2017 00:00:00
    25-Jul-2017 00:00:00    08-Aug-2017 00:00:00    10-Aug-2017 00:00:00    09-Sep-2017 00:00:00
    25-Apr-2017 00:00:00    08-May-2017 00:00:00    10-May-2017 00:00:00    10-Jun-2017 00:00:00
```

Display three declared dates. The `DeclaredDate` variable is a `datetime` array.

```
d.DVD_HIST{1}.DeclaredDate(1:3)
```

```
ans =

  3×1 datetime array

   31-Oct-2017 00:00:00
   25-Jul-2017 00:00:00
   25-Apr-2017 00:00:00
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

`f` — Bloomberg data fields
character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

`o` — Bloomberg override field
`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field name. A cell array of character vectors or string array denotes multiple Bloomberg override field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `'END_DT'`

Data Types: `char` | `cell` | `string`

`ov` — Bloomberg override field value
`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field value, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field value. A cell array of character vectors or string array denotes multiple Bloomberg override field values. Use this field value to filter the Bloomberg data result set.

Example: `'20100101'`

Data Types: `char` | `cell` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'returnFormattedValue',true`

`returnEids` — Entitlement identifiers
true | false

Entitlement identifiers, specified as the comma-separated pair consisting of `'returnEids'` and a Boolean. `true` adds a name and value for the entitlement identifier (EID) date to the return data.

Data Types: `logical`

`returnFormattedValue` — Return format
true | false

Return format, specified as the comma-separated pair consisting of `'returnFormattedValue'` and a Boolean. `true` forces all data to be returned as the data type character vector.

Data Types: `logical`

`useUTCTime` — Date time format
true | false

Date time format, specified as the comma-separated pair consisting of `'useUTCTime'` and a Boolean. `true` returns date and time values as Coordinated Universal Time (UTC) and `false` defaults to the Bloomberg **TZDF <GO>** settings of the requestor.

Data Types: `logical`

`forcedDelay` — Latest reference data
true | false

Latest reference data, specified as the comma-separated pair consisting of `'forcedDelay'` and a Boolean. `true` returns the latest data up to the delay period specified by the exchange for the security.

Data Types: `logical`

Output Arguments

`d` — Bloomberg data
structure (default) | table | timetable

Bloomberg data, returned as a structure, table, or timetable. The data type of the Bloomberg data depends on the DataReturnFormat and DatetimeType properties of the connection object. For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`sec` — Security list
cell array of character vectors

Security list, returned as a cell array of character vectors for the corresponding securities in `s`. The contents of `sec` are identical in value and order to `s`. You can return securities with any of the following identifiers:

- `buid`
- `cats`
- `cins`
- `common`
- `cusip`
- `isin`
- `sedol1`
- `sedol2`
- `sicovam`
- `svm`
- `ticker` (default)

- `wpk`

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface” on page 5-18
“Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15

getdata

Current data for Bloomberg Desktop connection V3

Syntax

```
d = getdata(c,s,f)
d = getdata(c,s,f,o,ov)
d = getdata(c,s,f,o,ov,Name,Value)
[d,sec] = getdata( ___ )
```

Description

`d = getdata(c,s,f)` returns the data for the fields `f` for the security list `s` using the `bloomberg` object `c` with the Bloomberg Desktop C++ interface. `getdata` accesses the Bloomberg reference data service.

`d = getdata(c,s,f,o,ov)` returns the data using the override fields `o` with corresponding override values `ov`.

`d = getdata(c,s,f,o,ov,Name,Value)` returns the data using name-value pair arguments for additional Bloomberg request settings.

`[d,sec] = getdata( ___ )` additionally returns the security list `sec` using any of the input argument combinations in the previous syntaxes.

Examples

Last and Open Price for Security

First, create a Bloomberg Desktop connection. Then, request last and open prices for a security. The current data you see when running this code can differ from the output data here.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Request last and open prices for Microsoft.

```
[d,sec] = getdata(c,'MSFT US Equity',{'LAST_PRICE';'OPEN'})
```

```
d =
    LAST_PRICE: 33.3401
          OPEN: 33.6000

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the security in `sec`.

Close the Bloomberg connection.

```
close(c)
```

Specified Fields Given Override Fields and Values

First, create a Bloomberg Desktop connection. Then, request data for specific fields for a security using an override field and value. The current data you see when running this code can differ from the output data here.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Request data for Bloomberg fields `'YLD_YTM_ASK'`, `'ASK'`, and `'OAS_SPREAD_ASK'` when the Bloomberg field `'OAS_VOL_ASK'` is `'14.000000'`.

```
[d,sec] = getdata(c,'030096AF8 Corp',...
   {'YLD_YTM_ASK','ASK','OAS_SPREAD_ASK','OAS_VOL_ASK'},...
   {'OAS_VOL_ASK'},{'14.000000'})
```

```
d =
       YLD_YTM_ASK: 5.6763
               ASK: 120.7500
    OAS_SPREAD_ASK: 307.9824
       OAS_VOL_ASK: 14

sec =
    '030096AF8 Corp'
```

`getdata` returns a structure `d` with the resulting values for the requested fields.

Close the Bloomberg connection.

```
close(c)
```

Request for Security Using CUSIP Number

First, create a Bloomberg Desktop connection. Then, use the CUSIP number for a security to request last price. The current data you see when running this code can differ from the output data here.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Request the last price for IBM with the CUSIP number.

```
d = getdata(c,'/cusip/459200101','LAST_PRICE')
```

```
d =
    LAST_PRICE: 182.5100
```

`getdata` returns a structure `d` with the last price.

Close the Bloomberg connection.

```
close(c)
```

Last Price for Security with Pricing Source

First, create a Bloomberg Desktop connection. Then, request the last price for a security. Specify the security using the CUSIP number with a pricing source. The current data you see when running this code can differ from the output data here.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Specify IBM with the CUSIP number and the pricing source `BGN` after the `@` symbol.

```
d = getdata(c,'/cusip/459200101@BGN','LAST_PRICE')
```

```
d =
    LAST_PRICE: 186.81
```

`getdata` returns a structure `d` with the last price.

Close the Bloomberg connection.

```
close(c)
```

Constituent Weights Using Date Override

First, create a Bloomberg Desktop connection. Then, request the constituent weights of an index using a date override. The current data you see when running this code can differ from the output data here.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return the constituent weights for the Dow Jones Index as of January 1, 2010, using a date override with the required date format `YYYYMMDD`.

```
d = getdata(c,'DJX Index','INDX_MWEIGHT','END_DT','20100101')
```

```
d =
    INDX_MWEIGHT: {{30x2 cell}}
```

`getdata` returns a structure `d` with a cell array where the first column is the index and the second column is the constituent weight.

Display the constituent weights for each index.

```
d.INDX_MWEIGHT{1,1}
```

```
ans =
    'AA UN'      [1.1683]
    'AXP UN'     [2.9366]
    'BA UN'      [3.9229]
```

```
    'BAC UN'     [1.0914]
    ...
```

Close the Bloomberg connection.

```
close(c)
```

Current Data and Dates as Table with Datetime

Create a Bloomberg connection, and then request current data for specific fields. The `getdata` function returns data for dates as a `datetime` array.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `getdata` function returns data as a structure.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Request current data for these fields:

- Last update date
- Last price
- Number of trades
- Previous real-time trading date

```
s = 'IBM US Equity';
f = {'LAST_UPDATE_DT','LAST_PRICE', ...
    'NUM_TRADES_RT','PREV_TRADING_DT_REALTIME'};
d = getdata(c,s,f)
```

```
d =

  1×4 table

      LAST_UPDATE_DT       LAST_PRICE    NUM_TRADES_RT    PREV_TRADING_DT_REALTIME
    ____________________    __________    _____________    ________________________

    21-Dec-2017 00:00:00      152.2           24846          20-Dec-2017 00:00:00
```

Display the last update date. This date is a `datetime` array.

```
d.LAST_UPDATE_DT
```

```
ans =
```

```
  datetime

   21-Dec-2017 00:00:00
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

f — Bloomberg data fields

character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

o — Bloomberg override field

`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field name. A cell array of character vectors or string array denotes multiple Bloomberg override field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `'END_DT'`

Data Types: `char` | `cell` | `string`

ov — Bloomberg override field value

`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field value, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field value. A

cell array of character vectors or string array denotes multiple Bloomberg override field values. Use this field value to filter the Bloomberg data result set.

Example: `'20100101'`

Data Types: `char` | `cell` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'returnEids',true`

`returnEids` — Entitlement identifiers
true | false

Entitlement identifiers, specified as the comma-separated pair consisting of `'returnEids'` and a Boolean. `true` adds a name and value for the entitlement identifier (EID) date to the return data.

Data Types: `logical`

`returnFormattedValue` — Return format
true | false

Return format, specified as the comma-separated pair consisting of `'returnFormattedValue'` and a Boolean. `true` forces all data to be returned as the data type character vector.

Data Types: `logical`

`useUTCTime` — Date time format
true | false

Date time format, specified as the comma-separated pair consisting of `'useUTCTime'` and a Boolean. `true` returns date and time values as Coordinated Universal Time (UTC) and `false` defaults to the Bloomberg **TZDF <GO>** settings of the requestor.

Data Types: `logical`

`forcedDelay` — Latest reference data
true | false

Latest reference data, specified as the comma-separated pair consisting of `'forcedDelay'` and a Boolean. `true` returns the latest data up to the delay period specified by the exchange for the security.

Data Types: `logical`

Output Arguments

`d` — Bloomberg data
structure (default) | table | timetable

Bloomberg data, returned as a structure, table, or timetable. The data type of the Bloomberg data depends on the DataReturnFormat and DatetimeType properties of the connection object. For details

about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`sec` — Security list
cell array of character vectors

Security list, returned as a cell array of character vectors for the corresponding securities in `s`. The contents of `sec` are identical in value and order to `s`. You can return securities with any of the following identifiers:

- `buid`
- `cats`
- `cins`
- `common`
- `cusip`
- `isin`
- `sedol1`
- `sedol2`
- `sicovam`
- `svm`
- `ticker` (default)
- `wpk`

Tips

- Bloomberg V3 data supports additional name-value pair arguments. To access further information on these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.
- You can check data and field availability by using the Bloomberg Excel Add-In.

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `history` | `realtime` | `timeseries`

Topics

"Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface" on page 5-18
"Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface" on page 5-15

history

Historical data for Bloomberg Desktop connection V3

Syntax

```
d = history(c,s,f,fromdate,todate)
d = history(c,s,f,fromdate,todate,period)
d = history(c,s,f,fromdate,todate,period,currency)
d = history(c,s,f,fromdate,todate,period,currency,Name,Value)
[d,sec] = history( ___ )
```

Description

`d = history(c,s,f,fromdate,todate)` returns the historical data for the security list `s` for the fields `f` for the dates `fromdate` through `todate` using the `bloomberg` object `c` with the Bloomberg Desktop C++ interface. Date strings can be input in any format recognized by MATLAB. `sec` is the security list that maps the order of the return data. The return data `d` is sorted to match the input order of `s`.

`d = history(c,s,f,fromdate,todate,period)` returns the historical data for the fields `f` and the dates `fromdate` through `todate` with a specific periodicity `period`.

`d = history(c,s,f,fromdate,todate,period,currency)` returns the historical data for the security list `s` for the fields `f` and the dates `fromdate` through `todate` based on the given currency `currency`.

`d = history(c,s,f,fromdate,todate,period,currency,Name,Value)` returns the historical data for the security list `s` using additional options specified by one or more name-value pair arguments.

`[d,sec] = history( ___ )` additionally returns the security list `sec` using any of the input argument combinations in the previous syntaxes. The return data, `d` and `sec`, are sorted to match the input order of `s`.

Examples

Daily Closing Prices Within Date Range

First, create a Bloomberg Desktop connection. Then, retrieve the daily closing price for a security within a date range.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Get the daily closing price from August 1, 2010, through August 10, 2010, for the IBM security.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','8/10/2010')
```

```
d =

     734352.00        123.55
     734353.00        123.18
     734354.00        124.03
     734355.00        124.56
     734356.00        123.58
     734359.00        125.34
     734360.00        125.19


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Monthly Closing Prices Within Date Range

First, create a Bloomberg Desktop connection. Then, retrieve the monthly closing prices for a security within a date range.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Get the monthly closing price from August 1, 2010, through December 10, 2010, for the IBM security.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','12/10/2010','monthly')

d =

     734360.00        125.19
     734391.00        121.53
     734421.00        131.85
     734452.00        139.78
     734482.00        138.13


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Monthly Closing Prices Within Date Range Using US Currency

First, create a Bloomberg Desktop connection. Then, retrieve the monthly closing prices for a security within a date range. Specify prices using the US currency.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Get the monthly closing price from August 1, 2010, through December 10, 2010, for the IBM security in US currency `'USD'`.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','12/10/2010','monthly','USD')
```

```
d =

     734360.00        125.19
     734391.00        121.53
     734421.00        131.85
     734452.00        139.78
     734482.00        138.13


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Monthly Closing Prices Within Date Range Using Currency with Specified Period

First, create a Bloomberg Desktop connection. Then, retrieve the monthly closing prices for a security within a date range. Specify prices using the US currency. Specify period values to customize the returned data.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Get the monthly closing price from August 1, 2010, through August 1, 2011, for the IBM security in US currency. The period values `'monthly'`, `'actual'`, and `'all_calendar_days'` specify returning actual monthly data for all calendar days. The period value `'nil_value'` specifies filling missing data values with a `NaN`.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','8/01/2011',{'monthly','actual',...
                  'all_calendar_days','nil_value'},'USD')
```

```
d =
```

```
      734351.00        128.40
      734382.00        125.77
      734412.00        135.64
      734443.00        143.32
      734473.00        144.41
      734504.00        146.76
      734535.00        163.56
      734563.00        159.97
      734594.00        164.27
      734624.00        170.58
      734655.00        166.56
      734685.00        174.54
      734716.00        180.75


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Daily Closing Prices Within Date Range Using Currency with Name-Value Pairs

First, create a Bloomberg Desktop connection. Then, retrieve the daily closing prices for a security within a date range. Specify prices using the US currency. Use name-value pair arguments to adjust the prices.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Get the daily closing price from August 1, 2010, through August 10, 2010, for the IBM security in U.S. currency `'USD'`. The prices are adjusted for normal cash and splits.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','8/10/2010','daily','USD',...
                  'adjustmentNormal',true,...
                  'adjustmentSplit',true)
```

```
d =

      734352.00        123.55
      734353.00        123.18
      734354.00        124.03
      734355.00        124.56
      734356.00        123.58
      734359.00        125.34
      734360.00        125.19


sec =
```

```
    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Daily Closing Prices Using CUSIP Number and Pricing Source

First, create a Bloomberg Desktop connection. Then, retrieve the daily closing prices for a security within a date range. Specify the security using the CUSIP number and a pricing source.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Get the daily closing price from January 1, 2012, through January 1, 2013, for the security specified with a CUSIP number `/cusip/459200101` and with pricing source `BGN`.

`d` contains the numeric representation for the date in the first column and the closing price in the second column.

```
d = history(c,'/cusip/459200101@BGN','LAST_PRICE',...
            '01/01/2012','01/01/2013')
```

```
d =

     734871.00        180.69
     734872.00        179.96
     734873.00        179.10
     ...
```

Close the Bloomberg connection.

```
close(c)
```

Closing Prices Within Date Range Using International Date Format

First, create a Bloomberg Desktop connection. Then, retrieve the closing prices for a security within a date range. Specify the dates for the range using an international date format.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return the closing price for the given dates in international format for the security `'MSFT@BGN US Equity'`.

```
stDt = datenum('01/06/11','dd/mm/yyyy');
endDt = datenum('01/06/12','dd/mm/yyyy');
[d,sec] = history(c,'MSFT@BGN US Equity','LAST_PRICE',...
                  stDt,endDt,{'previous_value','all_calendar_days'})
```

```
d =

     734655.00         22.92
     734656.00         22.72
     734657.00         22.42
     ...

sec =

    'MSFT@BGN US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Median Estimated Earnings Per Share Using Override Fields

First, create a Bloomberg Desktop connection. Then, retrieve the median earnings per share for a security within a date range. Specify an override field and value.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the median estimated earnings per share for AkzoNobel from October 1, 2010, through October 30, 2010. When specifying Bloomberg override fields, use the character vector `'overrideFields'`. The `overrideFields` argument must be an `n`-by-2 cell array, where the first column is the override field and the second column is the override value.

```
d = history(c,'AKZA NA Equity', ...
            'BEST_EPS_MEDIAN',datenum('01.10.2010', ...
            'dd.mm.yyyy'),datenum('30.10.2010','dd.mm.yyyy'), ...
            {'daily','calendar'},[],'overrideFields', ...
            {'BEST_FPERIOD_OVERRIDE','BF'})

d =

     734412.00          3.75
     734415.00          3.75
     734416.00          3.75
     ...
```

`d` returns the numeric representation for the date in the first column and the median estimated earnings per share in the second column.

Close the Bloomberg connection.

```
close(c)
```

Historical Data as Table with Dates

Create a Bloomberg connection, and then retrieve closing prices for a historical date range. The `history` function returns data for dates as a `datetime` array.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `history` function returns data as a numeric array.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve historical closing prices for IBM from August 1, 2010, through August 10, 2010. `d` is a table that contains dates as a `datetime` array.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE', ...
    '8/01/2010','8/10/2010')
```

```
d =

  7×2 table

       DATE        LAST_PRICE
    ___________    __________

    02-Aug-2010      130.76
    03-Aug-2010      130.37
    04-Aug-2010      131.27
    05-Aug-2010      131.83
    06-Aug-2010      130.14
    09-Aug-2010      132.00
    10-Aug-2010      131.84


sec =

  1×1 cell array

    {'IBM US Equity'}
```

Access dates in the returned data.

```
d.DATE
```

```
ans =
```

```
  7×1 datetime array

   02-Aug-2010
   03-Aug-2010
   04-Aug-2010
   05-Aug-2010
   06-Aug-2010
   09-Aug-2010
   10-Aug-2010
```

Close the Bloomberg connection.

```
close(c)
```

Historical Data as Timetable

Create a Bloomberg connection, and then retrieve closing prices for a historical date range. The `history` function returns data as a `timetable`.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return data as a `timetable` by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `history` function returns data as a numeric array.

```
c.DataReturnFormat = 'timetable';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve historical closing prices for IBM from August 1, 2010, through August 10, 2010. `d` is a `timetable` that contains dates in the first column.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE', ...
    '8/01/2010','8/10/2010')
```

```
d =

  7×1 timetable

      DATE        LAST_PRICE
    ____________    __________

    02-Aug-2010      130.76
    03-Aug-2010      130.37
    04-Aug-2010      131.27
    05-Aug-2010      131.83
    06-Aug-2010      130.14
    09-Aug-2010      132.00
    10-Aug-2010      131.84
```

```
sec =

  1×1 cell array

    {'IBM US Equity'}
```

Access dates in the returned data.

```
d.DATE
```

```
ans =

  7×1 datetime array

   02-Aug-2010
   03-Aug-2010
   04-Aug-2010
   05-Aug-2010
   06-Aug-2010
   09-Aug-2010
   10-Aug-2010
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

f — Bloomberg data fields

character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

`period` — Periodicity
`'daily'` | `'weekly'` | `'monthly'` | `'quarterly'` | ...

Periodicity, specified as one of these values to denote the data to return. For specifying multiple values, use a cell array. For example, when `period` is set to `{'daily','all_calendar_days'}`, `history` returns daily data for all calendar days, and reports missing data as `NaN`s. When `period` is set to `'active_days_only'`, `history` returns data using the default periodicity for active trading days only. The default periodicity depends on the security. If a security is reported on a monthly basis, the default periodicity is monthly. These tables show the values for `period`.

To specify the periodicity of the return data, see this table.

Value	Description
`'daily'`	Return data for each day.
`'weekly'`	Return data for each week.
`'monthly'`	Return data for each month.
`'quarterly'`	Return data for each quarter.
`'semi_annually'`	Return data semiannually.
`'yearly'`	Return data for each year.

The anchor date is the date to which all other reported dates are related. To specify the anchor date, see this table.

Value	Description
`'actual'`	Anchor date specification for an actual date. For this function, for periodicities other than daily, `todate` is the anchor date. If the period is weekly and the `todate` is a Thursday, every data point is a Thursday, or the nearest prior business day to Thursday. If the period is monthly and the `todate` is the 20th of a month, every data point is the 20th of each month in the date range.
`'calendar'`	Anchor date specification for a calendar year.
`'fiscal'`	Anchor date specification for a fiscal year.
`'none'`	Do not specify the anchor date.

To specify returning data for particular days, see this table.

Value	Description
`'non_trading_weekdays'`	Return data for all weekdays.
`'all_calendar_days'`	Return data for all calendar days.
`'active_days_only'`	Return data for only active trading days.

To specify how to fill missing values, see this table.

Value	Description
'previous_value'	Fill missing values with previous values for dates without trading activity for the security. If no previous value exists in the month before the fromdate, this function retains the missing values.
'nil_value'	Fill missing values with a NaN for dates without trading activity for the security.

Data Types: char | cell

currency — Currency
character vector | string scalar

Currency, specified as a character vector or string scalar to denote the ISO code for the currency of the returned data. For example, to specify output money values in U.S. currency, use USD for this argument.

Data Types: char | string

fromdate — Beginning date
double scalar | character vector | string scalar | datetime

Beginning date for the historical data, specified as a double scalar, character vector, string scalar, or datetime array. You can specify dates in any of the formats supported by datestr and datenum that show a year, month, and day.

Data Types: datetime | double | char | string

todate — End date
double scalar | character vector | string scalar | datetime

End date for the historical data, specified as a double scalar, character vector, string scalar, or datetime array. You can specify dates in any of the formats supported by datestr and datenum that show a year, month, and day.

Data Types: datetime | double | char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: 'adjustmentNormal',true

overrideFields — Override fields
cell array

Override fields, specified as the comma-separated pair consisting of 'overrideFields' and an n-by-2 cell array. The first column of the cell array is the override field and the second column is the override value.

Example: `'overrideFields', {'IVOL_DELTA_LEVEL','DELTA_LVL_10';'IVOL_DELTA_PUT_OR_CALL','IVOL_PUT';'IVOL_MATURITY','MATURITY_1STM'}`

Data Types: `cell`

`adjustmentNormal` — Historical normal pricing adjustment
true | false

Historical normal pricing adjustment, specified as the comma-separated pair consisting of `'adjustmentNormal'` and a Boolean to reflect:

- Regular Cash
- Interim
- 1st Interim
- 2nd Interim
- 3rd Interim
- 4th Interim
- 5th Interim
- Income
- Estimated
- Partnership Distribution
- Final
- Interest on Capital
- Distribution
- Prorated

For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

`adjustmentAbnormal` — Historical abnormal pricing adjustment
true | false

Historical abnormal pricing adjustment, specified as the comma-separated pair consisting of `'adjustmentAbnormal'` and a Boolean to reflect:

- Special Cash
- Liquidation
- Capital Gains
- Long-Term Capital Gains
- Short-Term Capital Gains
- Memorial
- Return of Capital
- Rights Redemption
- Miscellaneous

- Return Premium
- Preferred Rights Redemption
- Proceeds/Rights
- Proceeds/Shares
- Proceeds/Warrants

For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

`adjustmentSplit` — Historical split pricing or volume adjustment
true | false

Historical split pricing or volume adjustment, specified as the comma-separated pair consisting of `'adjustmentSplit'` and a Boolean to reflect:

- Spin-Offs
- Stock Splits/Consolidations
- Stock Dividend/Bonus
- Rights Offerings/Entitlement

For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

`adjustmentFollowDPDF` — Historical pricing adjustment
true (default) | false

Historical pricing adjustment, specified as the comma-separated pair consisting of `'adjustmentFollowDPDF'` and a Boolean. Setting this name-value pair follows the **DPDF <GO>** option from the Bloomberg terminal. For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

Output Arguments

`d` — Bloomberg historical data
numeric array (default) | table | timetable

Bloomberg historical data, returned as a numeric array, table, or timetable. The data type of the historical data depends on the DataReturnFormat and DatetimeType properties of the connection object. The first column (or field) in the historical data contains the date. The remaining columns contain the requested data fields.

For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`sec` — Security list
cell array of character vectors

Security list, returned as a cell array of character vectors for the corresponding securities in `s`. The contents of `sec` are identical in value and order to `s`. You can return securities with any of the following identifiers:

- `buid`
- `cats`
- `cins`
- `common`
- `cusip`
- `isin`
- `sedol1`
- `sedol2`
- `sicovam`
- `svm`
- `ticker` (default)
- `wpk`

Tips

- You can check data and field availability by using the Bloomberg Excel Add-In.

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `getdata` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Historical Data Using Bloomberg Desktop C++ Interface” on page 5-20
“Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15

isconnection

Determine Bloomberg Desktop connection V3

Syntax

```
v = isconnection(c)
```

Description

`v = isconnection(c)` returns `true` (`1`) if `c` is a valid Bloomberg V3 connection using the Bloomberg Desktop C++ interface and `false` (`0`) otherwise.

Examples

Validate the Bloomberg Connection

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `getdata` | `history` | `realtime` | `timeseries` | `fieldinfo` | `fieldsearch`

Topics

“Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15

lookup

Find information about securities for Bloomberg Desktop connection V3

Syntax

```
l = lookup(c,q,reqtype,Name,Value)
```

Description

`l = lookup(c,q,reqtype,Name,Value)` retrieves data based on criteria in the query `q` for a specific request type `reqtype` using the Bloomberg connection `c` with the Bloomberg Desktop C++ interface. For additional information about the query criteria and the possible name-value pair combinations, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Examples

Look Up Security

Create a Bloomberg connection, and then use the Security Lookup to retrieve information about the IBM corporate bond. For details about Bloomberg and the parameter values you can set, see the *Bloomberg API Developer's Guide* using the **WAPI<GO>** option from the Bloomberg terminal.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `lookup` function returns data as a structure.

```
c.DataReturnFormat = 'table';
```

Retrieve the instrument data for an IBM corporate bond with a maximum of 20 rows of data. The Security Lookup returns the security names and descriptions.

```
insts = lookup(c,'IBM','instrumentListRequest','maxResults',20, ...
    'yellowKeyFilter','YK_FILTER_CORP', ...
    'languageOverride','LANG_OVERRIDE_NONE');
```

Display the first three rows in the table. The first column contains the IBM corporate bond names, and the second column contains the bond descriptions.

```
insts(1:3,:)
```

```
ans =

  3×2 table

        security                      description
    ________________    ____________________________________________
```

```
    'DD103619 <corp>'    'International Business Machines Corp'
    '459200AG <corp>'    'International Business Machines Corp'
    'EC767659 <corp>'    'International Business Machines Corp'
```

Close the Bloomberg connection.

```
close(c)
```

Look Up Curve

Use the Curve Lookup to retrieve information about the `'GOLD'` related curve `'CD1016'`. For details about Bloomberg and the parameter values you can set, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the curve data for the credit default swap subtype of corporate bonds for a `'GOLD'` related curve `'CD1016'`. Return a maximum of `10` rows of data for the U.S. with `'USD'` currency.

```
curves = lookup(c,'GOLD','curveListRequest','maxResults',10,...
                'countryCode','US','currencyCode','USD',...
                'curveid','CD1016','type','CORP','subtype','CDS')
```

```
curves =

            curve: {'YCCD1016 Index'}
      description: {'Goldman Sachs Group Inc/The'}
          country: {'US'}
         currency: {'USD'}
          curveid: {'CD1016'}
             type: {'CORP'}
          subtype: {'CDS'}
        publisher: {'Bloomberg'}
            bbgid: {''}
```

One row of data displays as Bloomberg curve name `'YCCD1016 Index'` with Bloomberg description `'Goldman Sachs Group Inc/The'` in the U.S. with `'USD'` currency. The Bloomberg short-form identifier for the curve is `'CD1016'`. Bloomberg is the publisher and the `bbgid` is blank.

Close the Bloomberg connection.

```
close(c)
```

Look Up Government Security

Use the Government Security Lookup to retrieve information for United States Treasury bonds. For details about Bloomberg and the parameter values you can set, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Filter government security data with ticker filter of `'T'` for a maximum of `10` rows of data.

```
govts = lookup(c,'T','govtListRequest','maxResults',10,...
               'partialMatch',false)
```

```
govts =

      parseky: {10x1 cell}
         name: {10x1 cell}
       ticker: {10x1 cell}
```

The Government Security Lookup returns `parseky` data, the name, and ticker of the United States Treasury bonds.

Display the `parseky` data.

```
govts.parseky
```

```
ans =
    '912828VS Govt'
    '912828RE Govt'
    '912810RC Govt'
    '912810RB Govt'
    '912828VU Govt'
    '912828VV Govt'
    '912828VB Govt'
    '912828VR Govt'
    '912828VW Govt'
    '912828VQ Govt'
```

Display the names of the United States Treasury bonds.

```
govts.name
```

```
ans =
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
```

Display the tickers of the United States Treasury bonds.

```
govts.ticker
```

```
ans =
    'T'
    'T'
    'T'
    'T'
    'T'
    'T'
```

```
    'T'
    'T'
    'T'
    'T'
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

q — Keyword query

character vector | string scalar | cell array of character vectors | string array

Keyword query, specified as a character vector, string scalar, cell array of character vectors, or string array. Each character vector or string denotes an item for which information is requested. For example, the keyword query can be a security, a curve type, or a filter ticker.

Data Types: `char` | `cell` | `string`

reqtype — Request type

`'instrumentListRequest'` | `'curveListRequest'` | `'govtListRequest'`

Request type, specified as the preceding values to denote the type of information request. `'instrumentListRequest'` denotes a security or instrument lookup request. `'curveListRequest'` denotes a curve lookup request. `'govtListRequest'` denotes a government lookup request for government securities.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'maxResults'`, `20`, `'yellowKeyFilter'`, `'YK_FILTER_CORP'`, `'languageOverride'`, `'LANG_OVERRIDE_NONE'`, `'countryCode'`, `'US'`, `'currencyCode'`, `'USD'`, `'curveid'`, `'CD1016'`, `'type'`, `'CORP'`, `'subtype'`, `'CDS'`, `'partialMatch'`, `false`

maxResults — Number of rows in result data

numeric scalar

Number of rows in the result data, specified as the comma-separated pair consisting of `'maxResults'` and a numeric scalar to denote the total maximum number of rows of information to return. Result data can be one or more rows of data no greater than the number specified.

Data Types: `double`

yellowKeyFilter — Bloomberg yellow key filter

character vector | string scalar

Bloomberg yellow key filter, specified as the comma-separated pair consisting of `'yellowKeyFilter'` and a unique character vector or string scalar to denote the particular yellow key for government securities, corporate bonds, equities, and commodities, for example.

Data Types: `char` | `string`

`languageOverride` — Language override
character vector | string scalar

Language override, specified as the comma-separated pair consisting of `'languageOverride'` and a unique character vector or string scalar to denote a translation language for the result data.

Data Types: `char` | `string`

`countryCode` — Country code
character vector | string scalar

Country code, specified as the comma-separated pair consisting of `'countryCode'` and a character vector or string scalar to denote the country for the result data.

Data Types: `char` | `string`

`currencyCode` — Currency code
character vector | string scalar

Currency code, specified as the comma-separated pair consisting of `'currencyCode'` and a character vector or string scalar to denote the currency for the result data.

Data Types: `char` | `string`

`curveID` — Bloomberg short-form identifier for curve
character vector | string scalar

Bloomberg short-form identifier for a curve, specified as the comma-separated pair consisting of `'curveID'` and a character vector or string scalar.

Data Types: `char` | `string`

`type` — Bloomberg market sector type
character vector | string scalar

Bloomberg market sector type corresponding to the Bloomberg yellow keys, specified as the comma-separated pair consisting of `'type'` and a character vector or string scalar.

Data Types: `char` | `string`

`subtype` — Bloomberg market sector subtype
character vector | string scalar

Bloomberg market sector subtype, specified as the comma-separated pair consisting of `'subtype'` and a character vector or string scalar to further delineate the market sector type.

Data Types: `char` | `string`

`partialMatch` — Partial match on ticker
`true` | `false`

Partial match on ticker, specified as the comma-separated pair consisting of `'partialMatch'` and `true` or `false`. When set to `true`, you can filter securities by setting `q` to a query such as `'T*'`. When set to `false`, the securities are unfiltered.

Data Types: `logical`

Output Arguments

`l` — Lookup information

structure (default) | table

Lookup information, returned as a structure or table containing set properties depending on the request type. The data type of the lookup information depends on the DataReturnFormat property of the connection object.

For a list of the set properties and their descriptions, see the following tables.

'instrumentListRequest' Properties

Property	Description
`security`	Security name
`description`	Security long name

'curveListRequest' Properties

Property	Description
`curve`	Bloomberg curve name
`description`	Bloomberg description
`country`	Country code
`currency`	Currency code
`curveid`	Bloomberg short-form identifier for the curve
`type`	Bloomberg market sector type
`subtype`	Bloomberg market sector subtype
`publisher`	Bloomberg specified as publisher
`bbgid`	Bloomberg identifier

'govtListRequest' Properties

Property	Description
`parsekey`	Bloomberg security identifier (ticker or CUSIP, for example), price source, and source key (Bloomberg yellow key)
`name`	Government security name
`ticker`	Government security ticker

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface” on page 5-18
“Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15

portfolio

Current portfolio data for Bloomberg Desktop connection V3

Syntax

```
d = portfolio(c,p,f)
d = portfolio(c,p,f,o,ov)
[d,plist] = portfolio( ___ )
```

Description

`d = portfolio(c,p,f)` returns current portfolio data for the fields `f` in the portfolio `p` using the `bloomberg` object `c` with the Bloomberg Desktop C++ interface.

`d = portfolio(c,p,f,o,ov)` returns current portfolio data using override field `o` and override value `ov`.

`[d,plist] = portfolio( ___ )` also returns the portfolio list `plist` using any of the input argument combinations in the previous syntaxes.

Examples

Request Portfolio Data

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Request portfolio data for a custom portfolio with portfolio identifier `U335877-1 Client`. Request data using all fields `f`.

```
p = 'U335877-1 Client';
f = {'PORTFOLIO_MEMBERS','PORTFOLIO_MPOSITION',...
     'PORTFOLIO_MWEIGHT','PORTFOLIO_DATA'};

d = portfolio(c,p,f)

d =

    PORTFOLIO_MPOSITION: {{0x1 cell}}
      PORTFOLIO_MWEIGHT: {{0x1 cell}}
         PORTFOLIO_DATA: {{0x1 cell}}
      PORTFOLIO_MEMBERS: {{0x1 cell}}
```

`d` is a structure that contains portfolio data. Each structure field corresponds to data for each portfolio field.

Close the Bloomberg connection.

```
close(c)
```

Request Portfolio Data Using Specific Date

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Request portfolio data for a custom portfolio with portfolio identifier `U335877-1 Client`. Request data using all fields `f`. Filter the portfolio data by specifying the date of November 3, 2014, using the override value `REFERENCE_DATE` equal to `20141103`.

```
p = 'U335877-1 Client';
f = {'PORTFOLIO_MEMBERS','PORTFOLIO_MPOSITION',...
     'PORTFOLIO_MWEIGHT','PORTFOLIO_DATA'};
o = {'REFERENCE_DATE'};
ov = {'20141103'};

[d,plist] = portfolio(c,p,f,o,ov)
```

```
d =

    PORTFOLIO_MPOSITION: {{0x1 cell}}
      PORTFOLIO_MWEIGHT: {{0x1 cell}}
         PORTFOLIO_DATA: {{0x1 cell}}
      PORTFOLIO_MEMBERS: {{0x1 cell}}

plist =

    'U335877-1 Client'
```

`d` is a structure that contains portfolio data. Each structure field corresponds to data for each portfolio field.

`plist` is a cell array that contains the portfolio identifier.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

p — Portfolio

character vector | string scalar

Portfolio, specified as a character vector or string scalar. Specify the portfolio by the ID that you can find in the upper-right corner of the portfolio display page. Append the text `' Client'` (without quotes) to the ID. For example, if the ID is `U335877-1`, then specify `'U335877-1 Client'`.

Access the portfolio display page by using the **PRTU<GO>** option from the Bloomberg terminal. For details, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `'U335877-1 Client'`

Data Types: `char` | `cell` | `string`

f — Portfolio fields

`'PORTFOLIO_DATA'` | `'PORTFOLIO_MEMBERS'` | `'PORTFOLIO_MPOSITION'` | `'PORTFOLIO_MWEIGHT'`

Portfolio fields, specified as one of the preceding values for one field. To specify multiple fields, use a cell array of these values.

Bloomberg Field Name	Bloomberg Field Description
`'PORTFOLIO_DATA'`	Returns a list of the identifiers, positions, market values, cost, cost date, and cost foreign exchange rate of each security in a custom portfolio.
`'PORTFOLIO_MEMBERS'`	Returns a list of identifiers for the members of a custom portfolio.
`'PORTFOLIO_MPOSITION'`	Returns a list of identifiers and the position for each security in a custom portfolio.
`'PORTFOLIO_MWEIGHT'`	Returns a list of identifiers and the percentage weight for each security in a custom portfolio.

Data Types: `char` | `cell`

o — Bloomberg override field

character vector | string scalar | cell array of character vectors | string array

Bloomberg override field, specified as a character vector, string scalar, cell array of character vectors, or string array. The Bloomberg value `'REFERENCE_DATE'` denotes returning Bloomberg data for a specific date.

Data Types: `char` | `cell` | `string`

ov — Bloomberg override field value

`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field value, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field value. A cell array of character vectors or string array denotes multiple Bloomberg override field values. Use this field value to filter the Bloomberg data result set.

Example: `'20100101'`

Data Types: `char` | `cell` | `string`

Output Arguments

d — Portfolio data

structure (default) | table

Portfolio data, returned as a structure or table. The data type of the portfolio data depends on the DataReturnFormat property of the connection object.

`plist` — Portfolio list
cell array of character vectors

Portfolio list, returned as a cell array of character vectors for the corresponding portfolio identifiers in `p`. The contents of `plist` are identical in value and order to `p`.

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface” on page 5-18

realtime

Real-time data for Bloomberg Desktop connection V3

Syntax

```
d = realtime(c,s,f)
[~,t] = realtime(c,s,f,eventhandler)
```

Description

`d = realtime(c,s,f)` returns the data for the `bloomberg` object `c` with the Bloomberg Desktop C ++ interface, security list `s`, and requested fields `f`. `realtime` accesses the Bloomberg Market Data service.

`[~,t] = realtime(c,s,f,eventhandler)` returns an empty output and the timer `t` associated with the real-time event handler for the subscription list. Given connection `c`, the `realtime` function subscribes to a security or securities `s` and requests fields `f`, to update in real time while running an event handler `eventhandler`.

Examples

Retrieve Data for One Security

Retrieve a snapshot of data for one security only.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the last trade and volume of the IBM security.

```
d = realtime(c,'IBM US Equity',{'Last_Trade','Volume'})
```

```
d =

    LAST_TRADE: '181.76'
        VOLUME: '7277793'
```

Close the Bloomberg connection.

```
close(c)
```

Retrieve Data for One Security Using Event Handler

You can create your own event handler function to process Bloomberg data. For this example, use the event handler `disp` that displays Bloomberg stock tick data at the command line.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the last price and volume for the IBM security using the event handler `disp`.

```
[~,t] = realtime(c,'IBM US Equity',{'LAST_PRICE','VOLUME'}, ...
    'disp')
```

```
t =


   Timer Object: timer-4

   Timer Settings
      ExecutionMode: fixedRate
              Period: 0.05
            BusyMode: drop
             Running: off

   Callbacks
            TimerFcn: 1x5 cell array
            ErrorFcn: ''
            StartFcn: ''
             StopFcn: ''

   Columns 1 through 4

    {'SecurityID'   }    {'LAST_PRICE'}    {'SecurityID'   }    {'VOLUME'}
    {'IBM US Equity'}    {'118.490000'}    {'IBM US Equity'}    {'744066'}
...
```

`realtime` returns the MATLAB timer object with its properties. Then, `realtime` returns the stock tick data for the IBM security with the volume and last trade price.

Stop the display of real-time data.

```
stop(t)
c.Session.stopSubscriptions
```

Close the Bloomberg connection.

```
close(c)
```

Retrieve Data for Multiple Securities Using Event Handler

You can create your own event handler function to process Bloomberg data. For this example, use the event handler `disp` that displays Bloomberg stock tick data at the command line.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the last price and volume for IBM and Ford Motor Company securities.

```
[~,t] = realtime(c,{'IBM US Equity','F US Equity'}, ...
    {'LAST_PRICE','VOLUME'},'disp')
```

```
t =
```

```
   Timer Object: timer-4

   Timer Settings
      ExecutionMode: fixedRate
             Period: 0.05
          BusyMode: drop
           Running: off

   Callbacks
          TimerFcn: 1x5 cell array
          ErrorFcn: ''
          StartFcn: ''
           StopFcn: ''

Columns 1 through 6

    {'SecurityID' }    {'LAST_PRICE'}    {'SecurityID' }    {'VOLUME'  }    {'SecurityID'   }
    {'F US Equity'}    {'8.960000'  }    {'F US Equity'}    {'13423731'}    {'IBM US Equity'}

  Columns 7 through 8

    {'SecurityID'   }    {'VOLUME'}
    {'IBM US Equity'}    {'744066'}
...
```

`realtime` returns the MATLAB timer object with its properties. Then, `realtime` returns the stock tick data for the IBM and Ford Motor Company securities with the last price and volume.

Stop the display of real-time data.

```
stop(t)
c.Session.stopSubscriptions
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

f — Bloomberg data fields

character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

`eventhandler` — Event handler
character vector | string scalar

Event handler, specified as a character vector or string scalar that denotes the name of an event handler function that you define. You can define an event handler function to process any type of real-time Bloomberg events. The specified event handler function runs every time the timer fires.

Data Types: `char` | `string`

Output Arguments

`d` — Bloomberg data
structure (default) | table | timetable

Bloomberg data, returned as a structure, table, or timetable. The data type of the Bloomberg data depends on the DataReturnFormat and DatetimeType properties of the connection object. For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`t` — MATLAB timer
object

MATLAB timer, returned as a MATLAB object. For details about this object, see `timer`.

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `getdata` | `history` | `timeseries`

Topics

"Retrieve Bloomberg Real-Time Data Using Bloomberg Desktop C++ Interface" on page 5-26
"Writing and Running Custom Event Handler Functions" on page 1-26

tahistory

Historical technical analysis for Bloomberg Desktop connection V3

Syntax

```
d = tahistory(c)
d = tahistory(c,s,startdate,enddate,study,period,Name,Value)
```

Description

`d = tahistory(c)` returns the Bloomberg V3 session technical analysis data study and element definitions using the `bloomberg` object `c` with the Bloomberg Desktop C++ interface.

`d = tahistory(c,s,startdate,enddate,study,period,Name,Value)` returns the Bloomberg V3 session technical analysis data study and element definitions with additional options specified by one or more name-value pair arguments.

Examples

Request Bloomberg Directional Movement Indicator (DMI) Study for Security

Return all available Bloomberg studies and use the DMI study to run a technical analysis for a security.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

List the available Bloomberg studies.

```
d = tahistory(c)
```

```
d =

         dmiStudyAttributes: [1x1 struct]
       smavgStudyAttributes: [1x1 struct]
        bollStudyAttributes: [1x1 struct]
         maoStudyAttributes: [1x1 struct]
          fgStudyAttributes: [1x1 struct]
         rsiStudyAttributes: [1x1 struct]
        macdStudyAttributes: [1x1 struct]
         tasStudyAttributes: [1x1 struct]
       emavgStudyAttributes: [1x1 struct]
      maxminStudyAttributes: [1x1 struct]
        ptpsStudyAttributes: [1x1 struct]
        cmciStudyAttributes: [1x1 struct]
        wlprStudyAttributes: [1x1 struct]
       wmavgStudyAttributes: [1x1 struct]
     trenderStudyAttributes: [1x1 struct]
         gocStudyAttributes: [1x1 struct]
        kltnStudyAttributes: [1x1 struct]
```

```
    momentumStudyAttributes: [1x1 struct]
         rocStudyAttributes: [1x1 struct]
         maeStudyAttributes: [1x1 struct]
        hurstStudyAttributes: [1x1 struct]
         chkoStudyAttributes: [1x1 struct]
           teStudyAttributes: [1x1 struct]
        vmavgStudyAttributes: [1x1 struct]
        tmavgStudyAttributes: [1x1 struct]
          atrStudyAttributes: [1x1 struct]
          rexStudyAttributes: [1x1 struct]
          adoStudyAttributes: [1x1 struct]
           alStudyAttributes: [1x1 struct]
          etdStudyAttributes: [1x1 struct]
          vatStudyAttributes: [1x1 struct]
         tvatStudyAttributes: [1x1 struct]
           pdStudyAttributes: [1x1 struct]
           rvStudyAttributes: [1x1 struct]
      ipmavgStudyAttributes: [1x1 struct]
       pivotStudyAttributes: [1x1 struct]
          orStudyAttributes: [1x1 struct]
         pcrStudyAttributes: [1x1 struct]
          bsStudyAttributes: [1x1 struct]
```

`d` contains structures pertaining to each available Bloomberg study.

Display the name-value pairs for the DMI study.

```
d.dmiStudyAttributes
```

```
ans =

                period: [1x104 char]
     priceSourceHigh: [1x123 char]
      priceSourceLow: [1x121 char]
    priceSourceClose: [1x125 char]
```

Obtain more information about the `period` property.

```
d.dmiStudyAttributes.period
```

```
ans =

DEFINITION period {

    Min Value = 1

    Max Value = 1

    TYPE Int64

} // End Definition: period
```

Run the DMI study for the IBM security for the last month with `period` equal to `14`, the high price, the low price, and the closing price.

```
d = tahistory(c,'IBM US Equity',floor(now)-30,floor(now),'dmi',...
              'all_calendar_days','period',14,...
              'priceSourceHigh','PX_HIGH',...
              'priceSourceLow','PX_LOW','priceSourceClose','PX_LAST')
```

```
d =

         date: [31x1 double]
     DMI_PLUS: [31x1 double]
    DMI_MINUS: [31x1 double]
          ADX: [31x1 double]
         ADXR: [31x1 double]
```

`d` contains a `studyDataTable` with one `studyDataRow` for each interval returned.

Display the first five dates in the returned data.

```
d.date(1:5,1)
```

```
ans =

     735507.00
     735508.00
     735509.00
     735510.00
     735511.00
```

Display the first five prices in the plus DI line.

```
d.DMI_PLUS(1:5,1)
```

```
ans =

         18.92
         17.84
         16.83
         15.86
         15.63
```

Display the first five prices in the minus DI line.

```
d.DMI_MINUS(1:5,1)
```

```
ans =

         30.88
         29.12
         28.16
         30.67
         29.24
```

Display the first five values of the Average Directional Index.

```
d.ADX(1:5,1)
```

```
ans =

         22.15
         22.28
         22.49
         23.15
         23.67
```

Display the first five values of the Average Directional Movement Index Rating.

```
d.ADXR(1:5,1)

ans =

         25.20
         25.06
         25.05
         25.60
         26.30
```

Close the Bloomberg connection.

```
close(c)
```

Request DMI Study for Security with Pricing Source

Run a technical analysis to return the DMI study for a security with a pricing source.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Run the DMI study for the Microsoft security with pricing source ETPX for the last month with `period` equal to 14, the high price, the low price, and the closing price.

```
d = tahistory(c,'MSFT@ETPX US Equity',floor(now)-30,floor(now),...
             'dmi','all_calendar_days','period',14,...
             'priceSourceHigh','PX_HIGH','priceSourceLow','PX_LOW',...
             'priceSourceClose','PX_LAST')

d =

         date: [31x1 double]
     DMI_PLUS: [31x1 double]
    DMI_MINUS: [31x1 double]
          ADX: [31x1 double]
         ADXR: [31x1 double]
```

`d` contains a `studyDataTable` with one `studyDataRow` for each interval returned.

Display the first five dates in the returned data.

```
d.date(1:5,1)

ans =

     735507.00
     735508.00
     735509.00
     735510.00
     735511.00
```

Display the first five prices in the plus DI line.

```
d.DMI_PLUS(1:5,1)

ans =
```

```
         28.37
         30.63
         32.72
         30.65
         29.37
```

Display the first five prices in the minus DI line.

```
d.DMI_MINUS(1:5,1)
```

```
ans =

         21.97
         21.17
         19.47
         18.24
         17.48
```

Display the first values of the Average Directional Index.

```
d.ADX(1:5,1)
```

```
ans =

         13.53
         13.86
         14.69
         15.45
         16.16
```

Display the first five values of the Average Directional Movement Index Rating.

```
d.ADXR(1:5,1)
```

```
ans =

         15.45
         15.36
         15.53
         15.85
         16.37
```

Close the Bloomberg connection.

```
close(c)
```

Return DMI Study Data as Table with Dates

Create a Bloomberg connection, and then return data for a DMI study. The `tahistory` function returns data for dates as a `datetime` array.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `tahistory` function returns data as a structure.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Run the DMI study for the IBM® security from June 12, 2017, through June 16, 2017, with `period` equal to 14, the high price, the low price, and the closing price.

```
d = tahistory(c,'IBM US Equity','6/12/2017','6/16/2017','dmi', ...
    'all_calendar_days','period',14,'priceSourceHigh','PX_HIGH', ...
    'priceSourceLow','PX_LOW','priceSourceClose','PX_LAST');
```

Access the DMI study data for the first three dates.

```
d(1:3,:)
```

```
ans =

  3×5 table

      date        DMI_PLUS    DMI_MINUS     ADX     ADXR
    ___________    ________    _________    _____    _____

    12-Jun-2017     30.48        16.31      33.93    45.26
    13-Jun-2017     28.88        15.45      33.67    44.10
    14-Jun-2017     26.62        18.98      32.46    42.67
```

`d` is a `table` that contains these columns:

- `date` -- Date
- `DMI_PLUS` -- Prices in plus DI line
- `DMI_MINUS` -- Prices in minus DI line
- `ADX` -- Average Directional Index values
- `ADXR` -- Average Directional Movement Index Rating values

Access the first three dates in the returned data.

```
d.date(1:3)
```

```
ans =

  3×1 datetime array

   12-Jun-2017
   13-Jun-2017
   14-Jun-2017
```

Close the Bloomberg connection.

```
close(c)
```

Return DMI Study Data as Timetable

Create a Bloomberg connection, and then return data for a DMI study. The `tahistory` function returns data as a `timetable`.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `tahistory` function returns data as a structure.

```
c.DataReturnFormat = 'timetable';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Run the DMI study for the IBM® security from June 12, 2017, through June 16, 2017, with `period` equal to 14, the high price, the low price, and the closing price.

```
d = tahistory(c,'IBM US Equity','6/12/2017','6/16/2017','dmi', ...
    'all_calendar_days','period',14,'priceSourceHigh','PX_HIGH', ...
    'priceSourceLow','PX_LOW','priceSourceClose','PX_LAST');
```

Access the DMI study data for the first three dates.

```
d(1:3,:)
```

```
ans =

  3×4 timetable

       date        DMI_PLUS    DMI_MINUS     ADX     ADXR
    ___________    ________    _________    _____    _____

    12-Jun-2017     30.48         16.31      33.93    45.26
    13-Jun-2017     28.88         15.45      33.67    44.10
    14-Jun-2017     26.62         18.98      32.46    42.67
```

`d` is a `timetable` that contains these columns:

- `date` -- Date
- `DMI_PLUS` -- Prices in plus DI line
- `DMI_MINUS` -- Prices in minus DI line
- `ADX` -- Average Directional Index values
- `ADXR` -- Average Directional Movement Index Rating values

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

s — Security

character vector | string scalar

Security, specified as a character vector or string scalar for a single Bloomberg security.

Data Types: `char` | `string`

startdate — Start date

numeric scalar | character vector | string scalar

Start date, specified as a numeric scalar, character vector, or string scalar to denote the start date of the date range for the returned tick data.

Example: `floor(now-1)`

Data Types: `double` | `char` | `string`

enddate — End date

numeric scalar | character vector | string scalar

End date, specified as a numeric scalar, character vector, or string scalar to denote the end date of the date range for the returned tick data.

Example: `floor(now)`

Data Types: `double` | `char` | `string`

study — Study type

character vector | string scalar

Study type, specified as a character vector or string scalar to denote the study to use for historical analysis.

Data Types: `char` | `string`

period — Periodicity

`'daily'` | `'weekly'` | `'monthly'` | `'quarterly'` | ...

Periodicity, specified as one of these values to denote the data to return. For specifying multiple values, use a cell array. For example, when `period` is set to `{'daily','all_calendar_days'}`, `tahistory` returns daily data for all calendar days, and reports missing data as `NaN`s. When `period` is set to `'active_days_only'`, `tahistory` returns data using the default periodicity for active trading days only. The default periodicity depends on the security. If a security is reported on a monthly basis, the default periodicity is monthly. These tables show the values for `period`.

To specify the periodicity of the return data, see this table.

Value	Description
'daily'	Return data for each day.
'weekly'	Return data for each week.
'monthly'	Return data for each month.
'quarterly'	Return data for each quarter.
'semi_annually'	Return data semiannually.
'yearly'	Return data for each year.

The anchor date is the date to which all other reported dates are related. To specify the anchor date, see this table.

Value	Description
'actual'	Anchor date specification for an actual date. For this function, for periodicities other than daily, enddate is the anchor date. If the period is weekly and the enddate is a Thursday, every data point is a Thursday, or the nearest prior business day to Thursday. If the period is monthly and the enddate is the 20th of a month, every data point is the 20th of each month in the date range.
'calendar'	Anchor date specification for a calendar year.
'fiscal'	Anchor date specification for a fiscal year.

To specify returning data for particular days, see this table.

Value	Description
'non_trading_weekdays'	Return data for all weekdays.
'all_calendar_days'	Return data for all calendar days.
'active_days_only'	Return data for only active trading days.

To specify how to fill missing values, see this table.

Value	Description
'previous_value'	Fill missing values with previous values for dates without trading activity for the security.
'nil_value'	Fill missing values with a NaN for dates without trading activity for the security.

Data Types: char | cell

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'period',14`, `'priceSourceHigh','PX_HIGH'`, `'priceSourceLow','PX_LOW'`, `'priceSourceClose','PX_LAST'`

Note For details about the full list of name-value pair arguments, see the Bloomberg tool located at `C:\blp\API\APIv3\bin\BBAPIDemo.exe`.

`period` — Period
numeric scalar

Period, specified as the comma-separated pair consisting of `'period'` and a numeric scalar. For details about the period, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `double`

`priceSourceHigh` — High price
character vector | string scalar

High price, specified as the comma-separated pair consisting of `'priceSourceHigh'` and a character vector or string scalar. For details about the high price, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

`priceSourceLow` — Low price
character vector | string scalar

Low price, specified as the comma-separated pair consisting of `'priceSourceLow'` and a character vector or string scalar. For details about the low price, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

`priceSourceClose` — Closing price
character vector | string scalar

Closing price, specified as the comma-separated pair consisting of `'priceSourceClose'` and a character vector or string scalar. For details about the closing price, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

Output Arguments

`d` — Technical analysis data
structure (default) | table | timetable

Technical analysis data, returned as a structure, table, or timetable. The data type of the returned data depends on the DataReturnFormat and DatetimeType properties of the connection object.

For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Desktop C++ Interface” on page 5-18
“Retrieve Current and Historical Data Using Bloomberg Desktop C++ Interface” on page 5-15

timeseries

Intraday tick data for Bloomberg Desktop connection V3

Syntax

```
d = timeseries(c,s,date)
d = timeseries(c,s,date,interval,field)
d = timeseries(c,s,date,[],field,options,values)

d = timeseries(c,s,{startdate,enddate})
d = timeseries(c,s,{startdate,enddate},interval,field)
d = timeseries(c,s,{startdate,enddate},[],field)
d = timeseries(c,s,{startdate,enddate},[],field,options,values)

d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval)
d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval,field)

d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime},
interval)
d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime},
interval,field)
```

Description

`d = timeseries(c,s,date)` retrieves raw tick data using the `bloomberg` object `c` with the Bloomberg Desktop C++ interface and a security for a specific date.

`d = timeseries(c,s,date,interval,field)` retrieves raw tick data that is aggregated into intervals for a specific field.

`d = timeseries(c,s,date,[],field,options,values)` retrieves raw tick data without an aggregation interval for a specific field with the specified options and corresponding values.

`d = timeseries(c,s,{startdate,enddate})` retrieves raw tick data for a date range using a start date and an end date.

`d = timeseries(c,s,{startdate,enddate},interval,field)` retrieves raw tick data for a specific date range aggregated into intervals for a specific field.

`d = timeseries(c,s,{startdate,enddate},[],field)` retrieves raw tick data for a specific date range without an aggregation interval for a specific field.

`d = timeseries(c,s,{startdate,enddate},[],field,options,values)` retrieves raw tick data for a specific date range without an aggregation interval for a specific field with specified options and corresponding values.

`d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval)` retrieves raw trade tick data for a specific time range for each day within a specific date range, aggregated into intervals.

`d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval,field)` uses a specific field for tick data to return.

`d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime}, interval)` retrieves raw trade tick data for a whole day increment within a specific date and time range, aggregated into intervals.

`d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime}, interval,field)` uses a specific field for tick data to return.

Examples

Retrieve Tick Data for Specific Date and Pricing Source

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date. Use a security with and without a pricing source to retrieve tick data.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the trade tick series using the IBM security for today.

```
d = timeseries(c,'IBM US Equity',floor(now))
```

```
d =

    'TRADE'    [735537.40]    [181.69]    [100.00]
    'TRADE'    [735537.40]    [181.69]    [100.00]
    'TRADE'    [735537.40]    [181.68]    [100.00]
    ...
```

The columns in `d` are:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size

Here, the first row shows that 100 IBM shares sold for $181.69 today.

Retrieve the trade tick series using the Microsoft security with pricing source `ETPX` for today.

```
d = timeseries(c,'MSFT@ETPX US Equity',floor(now))
```

```
d =

    'TRADE'    [735537.40]    [35.53]    [100.00]
    'TRADE'    [735537.40]    [35.55]    [200.00]
    'TRADE'    [735537.40]    [35.55]    [100.00]
    ...
```

Here, the first row shows that 100 Microsoft shares are sold for $35.53 today.

Close the Bloomberg connection.

```
close(c)
```

Time Interval with Specific Field

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date. Specify the tick data to return using a time interval and field.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the trade tick series using the IBM security aggregated into 5-minute intervals for today.

```
d = timeseries(c,'IBM US Equity',floor(now),5,'Trade')
```

```
d =

  Columns 1 through 7

   735537.40        181.69        181.99        180.10        181.84     252322.00        861.00
   735537.40        181.90        181.97        181.57        181.65      78570.00        535.00
   735537.40        181.73        182.18        181.58        182.07     124898.00        817.00
     ...

  Column 8

   45815588.00
   14282076.00
   22710954.00
   ...
```

The columns in `d` contain the following:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

Here, the first row of data shows prices and tick data for the current date. The next row shows tick data for 5 minutes later.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Tick Data Using Option and Value

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date and field. Use option and value to return additional data.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the trade tick series using the `'F US Equity'` security without specifying the aggregation parameter for today. Also, return the condition codes.

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               'includeConditionCodes','true')
```

```
d =

    'TRADE'    [735556.57]    [17.12]    [   100.00]    'R6,IS'
    'TRADE'    [735556.57]    [17.12]    [   100.00]    ''
    'TRADE'    [735556.57]    [17.12]    [   500.00]    ''
    ...
```

The columns in `d` contain the following:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size
- Condition codes

Here, the first row shows that 100 `'F US Equity'` security shares sold for $17.12 today.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Tick Data Using Date Range

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the tick series for the `'F US Equity'` security for the last business day from the beginning of the day to noon.

```
d = timeseries(c,'F US Equity',{floor(now-4),floor(now-3.5)})
```

```
d =

    'TRADE'    [735552.67]    [17.09]    [   200.00]
    'TRADE'    [735552.67]    [17.09]    [   100.00]
    'TRADE'    [735552.67]    [17.09]    [   100.00]
   ...
```

The columns in `d` are:

- Tick type
- Numeric representation of the date and time

- Tick value
- Tick size

Here, the first row shows that 200 `'F US Equity'` security shares were sold for $17.09 on the last business day.

Close the Bloomberg connection.

```
close(c)
```

Date Range with Interval and Specific Field

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range. Specify the interval and field.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the trade tick series for the past 50 days for the IBM security aggregated into 5-minute intervals.

```
d = timeseries(c,'IBM US Equity',{floor(now)-50,floor(now)},5,'Trade')
```

```
ans =

  Columns 1 through 7

   735487.40        187.20        187.60        187.02        187.08     207683.00        560.00
   735487.40        187.03        187.13        186.65        186.78      46990.00        349.00
   735487.40        186.78        186.78        186.40        186.47      51589.00        399.00
     ...

Column 8

   38902968.00
    8779374.00
    9626896.00
   ...
```

The columns in `d` contain the following:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row of data shows prices and tick data for the current date. The next row shows tick data for 5 minutes later.

Close the Bloomberg connection.

```
close(c)
```

Date Range with Numerous Fields

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range and numerous fields.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return the Bid, Ask, and trade tick series for the security `'F US Equity'` for yesterday with a time interval at noon, without specifying the aggregation parameter.

```
d = timeseries(c,'F US Equity',{floor(now-1)+.5,floor(now-1)+.51},...
               [],{'Bid','Ask','Trade'})
```

```
d =

    'TRADE'    [735550.50]    [16.71]    [100.00]
    'ASK'      [735550.50]    [16.71]    [312.00]
    'BID'      [735550.50]    [16.70]    [177.00]
    ...
```

The columns in `d` are:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size

Here, the first row shows that 100 `'F US Equity'` security shares sold for $16.71 yesterday.

Close the Bloomberg connection.

```
close(c)
```

Date Range with Options and Values

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range. Specify options and values to return additional data.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return the trade tick series for the security `'F US Equity'` for yesterday with a time interval at noon, without specifying the aggregation parameter. Also, return the condition codes, exchange codes, and broker codes.

```
d = timeseries(c,'F US Equity',{floor(now-1)+.5,floor(now-1)+.51},...
               [],'Trade',{'includeConditionCodes',...
```

```
                'includeExchangeCodes','includeBrokerCodes'},...
                {'true','true','true'})

d =

    'TRADE'    [735550.50]    [16.71]    [100.00]    'T'     'D'
    'TRADE'    [735550.50]    [16.70]    [400.00]    'IS'    'B'
    'TRADE'    [735550.50]    [16.70]    [100.00]    'IS'    'B'
    ...
```

The columns in `d` contain the following:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size
- Exchange condition codes
- Exchange codes

Broker codes are available for Canadian, Finnish, Mexican, Philippine, and Swedish equities only. In this case, the broker buy code appears in the seventh column and the broker sell code appears in the eighth column.

Here, the first row shows that 100 `'F US Equity'` security shares sold for $16.71 yesterday.

Close the Bloomberg connection.

```
close(c)
```

Date and Time Range with Interval

Use Bloomberg to retrieve raw trade tick data by specifying a time range for each day in a specific date range. Specify the time interval for the tick data.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the trade tick series for the `'F US Equity'` security for the last two days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. `d` is a numeric matrix.

```
s = 'F US Equity';
startdate = datetime('today')-1;
enddate = datetime('today');
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;

d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

     736959.40         11.71         11.81         11.71         11.79
     736959.40         11.79         11.81         11.75         11.79
     736959.40         11.80         11.82         11.78         11.80

  Columns 6 through 8

    1375547.00       1190.00   16196757.00
     598924.00        898.00    7058724.00
     488655.00        641.00    5768371.50
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

Determine the maximum high price for the last two days.

```
highprices = d(:,3);
m = max(highprices)
```

```
m =

         11.82
```

Close the Bloomberg connection.

```
close(c)
```

Date and Time Range with Interval and Specific Field

Use Bloomberg to retrieve raw tick data by specifying a time range for each day in a specific date range. Specify the time interval and the field for the type of tick data to return. Here, specify the bid tick data.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the tick series for the `'F US Equity'` security for the last two days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. Specify retrieving the bid tick series. `d` is a numeric matrix.

```
s = 'F US Equity';
startdate = datetime('today')-1;
enddate = datetime('today');
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;
field = 'BID';

d = timeseries(c,s,{startdate:enddate,starttime,endtime}, ...
    interval,field);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    736959.40         11.70         11.80         11.70         11.79
    736959.40         11.79         11.80         11.75         11.79
    736959.40         11.79         11.81         11.78         11.80

  Columns 6 through 8

    397711.00       1442.00    4681704.50
    450997.00       1698.00    5311330.50
    464761.00       1391.00    5481707.50
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

Determine the maximum high price for the last two days.

```
highprices = d(:,3);
m = max(highprices)
```

```
m =

         11.81
```

Close the Bloomberg connection.

```
close(c)
```

Date and Time Range with Day Increment and Interval

Use Bloomberg to retrieve raw trade tick data by specifying a time range for each day in a specific date range. Specify a day increment for the date range and the time interval for the tick data.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the trade tick series for the `'IBM US Equity'` security for the last two months. Set the day increment to 5 days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. `d` is a numeric matrix.

```
s = 'IBM US Equity';
startdate = datetime('today')-60;
enddate = datetime('today');
dayincrement = 5;
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;

d = timeseries(c,s, ...
    {startdate:dayincrement:enddate,starttime,endtime}, ...
    interval);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    736900.40        147.00        147.04        146.55        146.62
    736900.40        146.62        146.87        146.62        146.71
    736900.40        146.72        146.79        146.52        146.54
```

```
  Columns 6 through 8

     125558.00        393.00   18440146.00
      39535.00        258.00    5800969.00
      49659.00        314.00    7282961.00
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

After the tick data for the first day in the date range, `d` contains tick data for a trading day that is 5 days later.

Close the Bloomberg connection.

```
close(c)
```

Date and Time Range with Day Increment, Interval, and Specific Field

Use Bloomberg to retrieve raw tick data by specifying a time range for each day in a specific date range. Specify a day increment for the date range, the time interval for the tick data, and the field for the type of tick data to return. Here, specify the bid tick data.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Retrieve the trade tick series for the `'F US Equity'` security for the last two months. Set the day increment to 5 days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. Specify the bid tick series. `d` is a numeric matrix.

```
s = 'F US Equity';
startdate = datetime('today')-60;
enddate = datetime('today');
dayincrement = 5;
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;
field = 'BID';
```

```
d = timeseries(c,s, ...
    {startdate:dayincrement:enddate,starttime,endtime}, ...
    interval,field);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

     736900.40         11.50         11.54         11.49         11.50
     736900.40         11.50         11.50         11.48         11.48
     736900.40         11.48         11.49         11.44         11.44

  Columns 6 through 8

     422305.00       1158.00    4863894.00
     575966.00       1180.00    6617854.00
     288147.00       1489.00    3305491.75
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

After the tick data for the first day in the date range, `d` contains tick data for a trading day that is 5 days later.

Close the Bloomberg connection.

```
close(c)
```

Return Tick Data as Table with Dates

Create a Bloomberg connection, and then return intraday tick data. The `timeseries` function returns data for dates as a `datetime` array.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `timeseries` function returns data as a numeric array.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve the trade tick series for the IBM® security aggregated into 5-minute intervals for today. `d` is a table that contains the tick series data.

```
s = 'IBM US Equity';
date = floor(now);
interval = 5;
field = 'Trade';

d = timeseries(c,s,date,interval,field);
```

Access the first three ticks of data.

```
d(1:3,:)
```

```
ans =

  3×8 table

      DATE         OPEN      HIGH      LOW      CLOSE      VOLUME      NUMBER_OF_TICKS    TOTAL_
    ___________    ______    ______    ______    ______    _________    _______________    ______

    21-Dec-2017    153.17    153.31    153.08    153.31    152524.00        442.00         23367
    21-Dec-2017    153.35    153.35    152.82    152.84     46051.00        291.00          7048
    21-Dec-2017    152.84    153.21    152.82    153.16     30966.00        225.00          4737
```

`d` contains columns with the following data:

- Date
- Open price
- High price
- Low price
- Closing price
- Volume
- Number of ticks
- Total tick value in the bar

Access the first three dates in the `DATE` column.

```
d.DATE(1:3)
```

```
ans =

  3×1 datetime array

   21-Dec-2017
   21-Dec-2017
   21-Dec-2017
```

Close the Bloomberg connection.

```
close(c)
```

Return Tick Data as Timetable

Create a Bloomberg connection, and then return intraday tick data. The `timeseries` function returns data for dates as a `timetable`.

Create the Bloomberg connection using the Bloomberg Desktop C++ interface.

```
c = bloomberg;
```

Return data as a `timetable` by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `timeseries` function returns data as a numeric array.

```
c.DataReturnFormat = 'timetable';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve the trade tick series for the IBM® security aggregated into 5-minute intervals for today. `d` is a `timetable` that contains the tick series data.

```
s = 'IBM US Equity';
date = floor(now);
interval = 5;
field = 'Trade';

d = timeseries(c,s,date,interval,field);
```

Access the first three ticks of data.

```
d(1:3,:)
```

```
ans =

  3×7 timetable

      DATE         OPEN      HIGH      LOW      CLOSE      VOLUME      NUMBER_OF_TICKS    TOTAL_
    ___________    ______    ______    ______    ______    _________    _______________    ______

    21-Dec-2017    153.17    153.31    153.08    153.31    152524.00        442.00         233670
```

```
    21-Dec-2017    153.35    153.35    152.82    152.84     46051.00        291.00          70486
    21-Dec-2017    152.84    153.21    152.82    153.16     30966.00        225.00          47373
```

`d` is a `timetable` that contains the following data:

- Date
- Open price
- High price
- Low price
- Closing price
- Volume
- Number of ticks
- Total tick value in the bar

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg connection

`bloomberg` object

Bloomberg connection, specified as a `bloomberg` object.

`s` — Security

character vector | string scalar

Security, specified as a character vector or string scalar for a single Bloomberg security.

Data Types: `char` | `string`

`date` — Date

numeric scalar | character vector | string scalar | `datetime` array

Date, specified as a numeric scalar, character vector, string scalar, or `datetime` array. `date` specifies the date for the returned tick data based on the entire day from midnight until 11:59:59 p.m.

Example: `floor(now)`

Data Types: `double` | `char` | `string` | `datetime`

`interval` — Time interval

numeric scalar

Time interval, specified as a numeric scalar to denote the number of minutes between ticks for the returned tick data.

Data Types: `double`

`field` — Bloomberg field

`'TRADE'` (default) | `'BID'` | `'ASK'` | ...

Bloomberg field, specified as one of these values that define the tick data to return.

Request Type	Valid Bloomberg Field Values
IntradayBarRequest with time interval specified	'TRADE'
	'BID'
	'ASK'
	'BID_BEST'
	'ASK_BEST'
IntradayTickRequest with no time interval specified	'TRADE'
	'BID'
	'ASK'
	'BID_BEST'
	'ASK_BEST'
	'SETTLE'

`options` — Bloomberg API options
`'includeConditionCodes'` | `'includeExchangeCodes'` | `'includeBrokerCodes'` | ...

Bloomberg API options, specified as one of the values in this table.

Value	Description
'includeConditionCodes'	Exchange condition codes associated with the event
'includeExchangeCodes'	Exchange code where tick originated
'includeBrokerCodes'	Broker code
'includeRpsCodes'	Reporting party side
'includeNonPlottableEvents'	After-hours data

Note The value `'includeNonPlottableEvents'` applies to raw intraday requests only.

To specify more than one Bloomberg API option, use a cell array of these values.

Specify the corresponding Bloomberg API value for each API option. The number of options must match the number of values.

For example, to specify one Bloomberg API option, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               'includeConditionCodes','true');
```

To specify two Bloomberg API options, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               {'includeConditionCodes','includeExchangeCodes'},...
               {'true','true'});
```

For details about the options, see the *Bloomberg API Developer's Guide*.

Data Types: `char` | `cell`

values — Bloomberg API values
'true' | 'false'

Bloomberg API values, specified as 'true' or 'false'. Each value corresponds to the specified Bloomberg API option. To specify more than one Bloomberg API value, use a cell array. The number of values must match the number of options.

For example, to specify one Bloomberg API option, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               'includeConditionCodes','true');
```

To specify two Bloomberg API options, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               {'includeConditionCodes','includeExchangeCodes'},...
               {'true','true'});
```

Data Types: char | cell

startdate — Start date
numeric scalar | character vector | string scalar | datetime array

Start date, specified as a numeric scalar, character vector, string scalar, or datetime array. This date specifies the beginning of the date range for the returned tick data. If no ticks are present in the date range, then returned tick data is empty.

Example: floor(now-1)

Data Types: double | char | string | datetime

enddate — End date
numeric scalar | character vector | string scalar | datetime array

End date, specified as a numeric scalar, character vector, string scalar, or datetime array. This date specifies the end of the date range for the returned tick data. If no ticks are present in the date range, then returned tick data is empty.

Example: floor(now)

Data Types: double | char | string | datetime

starttime — Start time
character vector | string scalar | datetime array

Start time, specified as a character vector, string scalar, or datetime array. This time specifies the start time of the time range for the returned tick data.

Example: '09:30:00'

Data Types: char | string | datetime

endtime — End time
character vector | string scalar | datetime array

End time, specified as a character vector, string scalar, or datetime array. This time specifies the end time of the time range for the returned tick data.

Example: '16:30:00'

Data Types: `char` | `string` | `datetime`

`dayincrement` — Day increment
1 (default) | numeric scalar

Day increment, specified as a numeric scalar. This number specifies the whole day increment for a specific date range. For example, if the day increment is 7, then the returned data contains ticks for every 7th day starting from the first day within the date range.

Data Types: `double`

Output Arguments

`d` — Bloomberg tick data
cell array | numeric array | table | timetable

Bloomberg tick data, returned as one of these data types:

- Cell array for requests without a specified time interval (raw tick data)
- Numeric array for requests with a specified time interval
- table
- timetable

The data type of the tick data depends on the DataReturnFormat and DatetimeType properties of the connection object.

Note The Bloomberg API returns the tick time with precision in seconds.

Limitations

When the data request is too large, `timeseries` displays this error message:

```
Timeout error:
Error using blp/timeseries>processResponseEvent (line 338) REQUEST FAILED: responseError = {

source = bbdbl7

code = -2

category = TIMEOUT

message = Timed out getting data from store [nid:327]

subcategory = INTERNAL_ERROR

}
```

To fix this error, shorten the length of the date range by modifying the input arguments `startdate` and `enddate`.

Tips

- You cannot retrieve Bloomberg intraday tick data for a date more than 140 days ago.
- The *Bloomberg API Developer's Guide* states that `'TRADE'` corresponds to LAST_PRICE for IntradayTickRequest and IntradayBarRequest.

- Bloomberg V3 intraday tick data supports additional name-value pairs. For details on these pairs, see the *Bloomberg API Developer's Guide* by typing `WAPI` and clicking the **<GO>** button on the Bloomberg terminal.
- You can check data and field availability by using the Bloomberg Excel Add-In.

Version History

Introduced in R2021a

See Also

`bloomberg` | `close` | `getdata` | `history` | `realtime`

Topics

"Retrieve Bloomberg Intraday Tick Data Using Bloomberg Desktop C++ Interface" on page 5-24

bloombergBPIPE

Bloomberg B-PIPE connection V3

Description

The `bloombergBPIPE` function creates a `bloombergBPIPE` object. The `bloombergBPIPE` object represents a Bloomberg B-PIPE connection using the Bloomberg V3 C++ API.

Other Datafeed Toolbox functions connect to different Bloomberg services: Bloomberg Desktop (`bloomberg`) and Bloomberg Server (`bloombergServer`). For details about these services, see “Comparing Bloomberg Connections” on page 2-4.

For details about Bloomberg connection requirements, see “Data Server Connection Requirements” on page 1-3. To ensure a successful Bloomberg connection, perform the required steps before executing `bloombergBPIPE`. For details, see “Installing Bloomberg and Configuring Connections” on page 1-5.

Creation

Syntax

```
c = bloombergBPIPE(authtype,appname,ipaddress,port)
c = bloombergBPIPE(authtype,appname,ipaddress,port,timeout)
```

Description

`c = bloombergBPIPE(authtype,appname,ipaddress,port)` creates a Bloomberg B-PIPE connection object `c`, and sets these properties:

- authtype
- appname
- ipaddress
- port

`c = bloombergBPIPE(authtype,appname,ipaddress,port,timeout)` also sets the timeout property.

Caution To refer to a Bloomberg connection in other functions, use the connection object created by the `bloombergBPIPE` function. Otherwise, using `bloombergBPIPE` as an input argument opens multiple Bloomberg connections, causing unexpected behavior and exhausting memory resources.

Properties

`AppAuthType` — Application authentication type
`""` (default) | `"APPNAME_AND_KEY"`

This property is read-only.

Application authentication type, specified as one of these values:

- `""` — Bloomberg B-PIPE connection with Windows authentication
- `"APPNAME_AND_KEY"` — Bloomberg B-PIPE connection with application authentication

`AuthType` — Bloomberg user authentication type
`"OS_LOGON"` | `"APPLICATION_ONLY"`

Bloomberg user authentication type, specified as one of these values:

- `"OS_LOGON"` — Bloomberg B-PIPE connection with Windows authentication
- `"APPLICATION_ONLY"` — Bloomberg B-PIPE connection with application authentication

For details, see the *Bloomberg B-PIPE API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`AppName` — Application name
character vector | string

Application name, specified as a character vector or string that identifies the application you are using to connect to Bloomberg B-PIPE.

Example: `'appname'`

Data Types: `char` | `string`

`User` — Bloomberg user
Bloomberg user identity object

This property is read-only.

Bloomberg user, specified as a Bloomberg user identity object.

Example: `[1x1 com.bloomberglp.blpapi.impl.aT]`

`Session` — Bloomberg V3 session
Bloomberg V3 API Session object

This property is read-only.

Bloomberg V3 session, specified as a Bloomberg V3 API Session object.

Example: `[1x1 BLPSession]`

`IPAddress` — IP address
character vector | cell array of character vectors | string | string array

IP address of the machine running the Bloomberg B-PIPE process, specified as a character vector, cell array of character vectors, string, or string array. A character vector or string identifies the machine running the Bloomberg B-PIPE process, whereas a cell array of character vectors or string array specifies multiple machines.

Example: `{'111.11.11.112'}`

Data Types: `char` | `cell` | `string`

`Port` — Port number

`[]` (default) | numeric scalar

Port number of the machine running the Bloomberg B-PIPE process, specified as a numeric scalar.

Example: `8194`

Data Types: `double`

`TimeOut` — Timeout

numeric scalar

Timeout specifying the time in milliseconds that MATLAB attempts to connect to the machine running the Bloomberg B-PIPE process before timing out, specified as a numeric scalar.

Example: `1000`

Data Types: `double`

`DatetimeType` — Date and time data type

`''` (default) | `'datetime'`

Date and time data type, specified as one of these values.

Value	Description
`''` (default)	Return date and time values as MATLAB date numbers.
`'datetime'`	Return date and time values as a `datetime` array.

You can specify these values using a character vector or string (for example, `"datetime"`).

When you create a `bloombergBPIPE` object, the `bloombergBPIPE` function leaves this property unset. To retrieve data, you must set this property value manually at the command line or in a script using dot notation, for example:

```
c.DatetimeType = 'datetime';
```

Then, you can use these supported functions:

- getbulkdata
- getdata
- history
- tahistory
- timeseries

Note If the `DataReturnFormat` property value is `'table'` and the `DatetimeType` property value is `'datetime'`, then the returned data is a table that contains date and time values as a `datetime` array. If the `DataReturnFormat` property value is an empty character vector, then setting the `DatetimeType` property to `'datetime'` returns date and time values for aggregated ticks and historical requests as MATLAB date numbers.

`DataReturnFormat` — Data return format
`'cell'` | `'structure'` | `'table'` | `'timetable'`

Data return format, specified as one of these values, which determine the data type of the returned data.

Value	Data Type of Returned Data
`'cell'`	`cell array`
`'table'`	`table`
`'timetable'`	`timetable`
`'structure'`	`structure`

Note The default data type of the returned data depends on the executed function. To specify the default data type, set the `DataReturnFormat` property to `''`. For default data types, see the supported function list.

You can specify these values using a character vector or string (for example, `"table"`).

When you create a `bloombergBPIPE` object, the `bloombergBPIPE` function leaves this property unset. To retrieve data, you must set this property value manually at the command line or in a script using dot notation, for example:

```
c.DataReturnFormat = 'structure';
```

Then, you can use these supported functions.

Supported Function	Valid Data Types for Returned Data
category	• `cell array` (default) • `structure` • `table`
eqs	• `cell array` (default) • `structure` • `table`
fieldinfo	• `cell array` (default) • `structure` • `table`

Supported Function	Valid Data Types for Returned Data
fieldsearch	• `cell array` (default) • `structure` • `table`
lookup	• `structure` (default) • `table`
portfolio	• `structure` (default) • `table`
getbulkdata	• `structure` (default) • `table` • `timetable`
getdata	• `structure` (default) • `table` • `timetable`
history	• numeric array (default) • `table` • `timetable`
tahistory	• `structure` (default) • `table` • `timetable`
timeseries	• `cell array` (default for raw tick data) • numeric array (default for interval tick data) • `table` • `timetable`

Note Regardless of the `DatetimeType` property value, if the `DataReturnFormat` property value is `'timetable'`, then the `getdata` and `getbulkdata` functions return a table that contains date and time values as `datetime` arrays.

Object Functions

Bloomberg B-PIPE Connection

close Close Bloomberg B-PIPE connection V3
isconnection Determine Bloomberg B-PIPE connection V3

Bloomberg B-PIPE Data Retrieval

eqs Equity screening data for Bloomberg B-PIPE connection V3

get	Properties of Bloomberg B-PIPE connection V3
getbulkdata	Bulk data with header information for Bloomberg B-PIPE connection V3
getdata	Current data for Bloomberg B-PIPE connection V3
history	Historical data for Bloomberg B-PIPE connection V3
portfolio	Current portfolio data for Bloomberg B-PIPE connection V3
realtime	Real-time data for Bloomberg B-PIPE connection V3
tahistory	Historical technical analysis for Bloomberg B-PIPE connection V3
timeseries	Intraday tick data for Bloomberg B-PIPE connection V3

Retrieve Bloomberg B-PIPE Information

category	Field category search for Bloomberg B-PIPE connection V3
fieldinfo	Field information for Bloomberg B-PIPE connection V3
fieldsearch	Field search for Bloomberg B-PIPE connection V3
lookup	Find information about securities for Bloomberg B-PIPE connection V3

Examples

Create Bloomberg B-PIPE Connection

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port)
```

```
c =

  bloombergBPIPE with properties:

         AppAuthType: ''
            AuthType: 'OS_LOGON'
             AppName: []
                User: []
             Session: [1×1 BLPSession]
           IPAddress: {'111.11.11.112'}
                Port: 8194.00
             TimeOut: 0
        DatetimeType: ''
    DataReturnFormat: ''
```

The `bloombergBPIPE` function connects to the machine running Bloomberg B-PIPE at port number `8194`. The `bloombergBPIPE` function creates the `bloombergBPIPE` object `c` with these properties:

- Application authentication type
- Bloomberg user authentication type
- Application name
- Bloomberg user identity object
- Bloomberg V3 API Session object
- IP address of the machine running the Bloomberg B-PIPE process
- Port number of the machine running the Bloomberg B-PIPE process
- Number (in milliseconds) specifying how long MATLAB attempts to connect to the machine before timing out
- Date and time data type
- Data return format

Request the last and open prices for Microsoft.

```
format bank  % Display data format for currency
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
[d,sec] = getdata(c,s,f)
```

```
d =
    LAST_PRICE: 33.34
          OPEN: 33.60

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the name of the security in `sec`.

Close the Bloomberg B-PIPE connection.

```
close(c)
```

Create Bloomberg B-PIPE Connection with Timeout

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.
- The timeout value is 1000 milliseconds.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;
timeout = 1000;
```

```
c = bloombergBPIPE(authtype,appname,ipaddress,port,timeout)

c =

  bloombergBPIPE with properties:

         AppAuthType: ''
            AuthType: 'OS_LOGON'
             AppName: []
                User: []
             Session: [1×1 BLPSession]
           IPAddress: {'172.28.17.118'}
                Port: 8194.00
             TimeOut: 1000.00
        DatetimeType: ''
    DataReturnFormat: ''
```

The `bloombergBPIPE` function connects to the machine running Bloomberg B-PIPE at port number `8194`. The `bloombergBPIPE` function creates the `bloombergBPIPE` object `c` with these properties:

- Application authentication type
- Bloomberg user authentication type
- Application name
- Bloomberg user identity object
- Bloomberg V3 API Session object
- IP address of the machine running the Bloomberg B-PIPE process
- Port number of the machine running the Bloomberg B-PIPE process
- Number (in milliseconds) specifying how long MATLAB attempts to connect to the machine before timing out
- Date and time data type
- Data return format

Request the last and open prices for Microsoft.

```
format bank  % Display data format for currency
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
[d,sec] = getdata(c,s,f)

d =
    LAST_PRICE: 33.34
          OPEN: 33.60

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the name of the security in `sec`.

Close the Bloomberg B-PIPE connection.

```
close(c)
```

Version History

Introduced in R2021a

See Also

Topics

“Data Server Connection Requirements” on page 1-3
“Comparing Bloomberg Connections” on page 2-4
“Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface” on page 5-28
“Retrieve Bloomberg Historical Data Using Bloomberg B-PIPE C++ Interface” on page 5-31

category

Field category search for Bloomberg B-PIPE connection V3

Syntax

```
d = category(c,f)
```

Description

`d = category(c,f)` returns category information given the search term `f` using the Bloomberg B-PIPE C++ interface.

Examples

Search for Bloomberg Last Price Field

Create a Bloomberg connection, and then request the category description of the last price field.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return data as a table by setting the `DataReturnFormat` property of the `bloomberg` object. If you do not set this property, the `category` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Request the Bloomberg category description of the last price field.

```
f = 'LAST_PRICE';
d = category(c,f);
```

Display the first three rows of the Bloomberg category description data in `d`.

```
d(1:3,:)
```

```
ans =

  3×5 table

    CATEGORY       ID                  MNEMONIC                          DESCRIPTION
    __________    _______    ____________________________________    ____________________________________

    'Analysis'    'OP179'    'THETA_LAST'                           'Theta Last Price'
    'Analysis'    'VM048'    'DDMX_PERCENT_CHANGE_LAST_PRICE'       'DDMX Percent Change Last Price
    'Analysis'    'YL005'    'YLD_CNV_LAST'                         'Last Yield To Convention'
```

The columns in `d` are:

- Category
- Field identifier
- Field mnemonic
- Field name
- Field data type

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

f — Search term

character vector | string scalar

Search term, specified as a character vector or string scalar to denote Bloomberg fields.

Data Types: `char` | `string`

Output Arguments

d — Category data

cell array (default) | structure | table

Category data, returned as an N-by-5 cell array, a structure, or a table.

The columns (or fields) of the data types are:

- Category
- Field identifier
- Field mnemonic
- Field name

- Field data type

The data type of the category data depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `getdata` | `history` | `realtime` | `timeseries` | `fieldinfo` | `fieldsearch`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface” on page 5-28

close

Close Bloomberg B-PIPE connection V3

Syntax

```
close(c)
```

Description

`close(c)` closes the Bloomberg B-PIPE connection V3 `c` using the Bloomberg B-PIPE C++ interface.

Examples

Close Bloomberg Connection

First, create a Bloomberg B-PIPE connection. Then, request last and open prices for a security and close the connection. The current data you see when running this code can differ from the output data here.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Request last and open prices for Microsoft.

```
[d,sec] = getdata(c,'MSFT US Equity',{'LAST_PRICE';'OPEN'})
```

```
d =
    LAST_PRICE: 33.3401
          OPEN: 33.6000

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the security in `sec`.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface” on page 5-28
“Retrieve Bloomberg Historical Data Using Bloomberg B-PIPE C++ Interface” on page 5-31
“Retrieve Bloomberg Intraday Tick Data Using Bloomberg B-PIPE C++ Interface” on page 5-35
“Retrieve Bloomberg Real-Time Data Using Bloomberg B-PIPE C++ Interface” on page 5-37

eqs

Equity screening data for Bloomberg B-PIPE connection V3

Syntax

```
d = eqs(c,sname)
d = eqs(c,sname,stype)
d = eqs(c,sname,stype,languageid)
d = eqs(c,sname,stype,languageid,group)
d = eqs(c,sname,stype,languageid,group,'OverrideFields',ov)
```

Description

`d = eqs(c,sname)` returns equity screening data given the Bloomberg B-PIPE V3 session screen name `sname`.

`d = eqs(c,sname,stype)` also specifies the screen type `stype`.

`d = eqs(c,sname,stype,languageid)` also specifies the language identifier `languageid`.

`d = eqs(c,sname,stype,languageid,group)` also specifies the optional group identifier `group`.

`d = eqs(c,sname,stype,languageid,group,'OverrideFields',ov)` also specifies the Bloomberg override fields and values `ov`.

Examples

Retrieve Equity Screening Data for Screen

Create a Bloomberg connection, and then retrieve frontier market stock data.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return data as a table by setting the `DataReturnFormat` property of the `bloomberg` object. If you do not set this property, the `eqs` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Retrieve equity screening data for the screen named `Frontier Market Stocks with 1 billion USD Market Caps`.

```
sname = 'Frontier Market Stocks with 1 billion USD Market Caps';
d = eqs(c,sname);
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  3×8 table

      Cntry                 Name                    IndGroup    MarketCap     Price_D_1     P_B
    ___________    ___________________________    ________    __________    _________    _____

    'Venezuela'    'MERCANTIL SERVICIOS FINAN-A'    'Banks'     7.3424e+12      70088      278.25
    'Venezuela'    'BANCO DEL CARIBE-A'             'Banks'     2.0442e+12      24531      2321.8
    'Venezuela'    'BANCO PROVINCIAL'               'Banks'     1.2632e+12      11715      52.344
```

The columns in d are:

- Country name
- Company name
- Industry name
- Market capitalization
- Price
- Price-to-book ratio
- Price-earnings ratio
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Retrieve Equity Screening Data for a Screen Type

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.

- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve equity screening data for the screen called `Vehicle-Engine-Parts` and the screen type equal to `'GLOBAL'`.

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL');
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    'Ticker'        'Short Name'        'Market Cap'        'Price:D-1'    'P/E'
    'HON     US'    'HONEYWELL INTL'    [69451382784.00]    [    88.51]    [16.81]
    'CMI     US'    'CUMMINS INC'       [24799526912.00]    [   132.36]    [17.28]

  Columns 6 through 8

    'Total Return YTD'    'Revenue T12M'      'EPS T12M'
    [           42.43]    [38248998912.00]    [    4.11]
    [           24.43]    [17004999936.00]    [    7.57]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen. The first row contains column headers. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Retrieve Equity Screening Data for a Screen in German

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve equity screening data for the screen called `Vehicle-Engine-Parts`, the screen type equal to `'GLOBAL'`, and return data in German.

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL','GERMAN');
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
  Columns 1 through 5

    'Ticker'        'Kurzname'          'Marktkapitalisie...'    'Preis:D-1'    'KGV'
    'HON     US'    'HONEYWELL INTL'    [     69451382784.00]    [    88.51]    [16.81]
    'CMI     US'    'CUMMINS INC'       [     24799526912.00]    [   132.36]    [17.28]

  Columns 6 through 8

    'Gesamtertrag YTD'    'Erlös T12M'        'EPS T12M'
    [           42.43]    [38248998912.00]    [    4.11]
    [           24.43]    [17004999936.00]    [    7.57]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen. The first row contains column headers in German. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Retrieve Equity Screening Data for a Screen with a Specified Screen Folder Name

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve equity screening data for the Bloomberg screen called `Vehicle-Engine-Parts`, using the Bloomberg screen type `'GLOBAL'` and the language `'ENGLISH'`, and the Bloomberg screen folder name `'GENERAL'`.

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL','ENGLISH','GENERAL');
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    'Ticker'        'Short Name'        'Market Cap'        'Price:D-1'    'P/E'
    'HON     US'    'HONEYWELL INTL'    [69451382784.00]    [    88.51]    [16.81]
    'CMI     US'    'CUMMINS INC'       [24799526912.00]    [   132.36]    [17.28]

  Columns 6 through 8

    'Total Return YTD'    'Revenue T12M'      'EPS T12M'
    [           42.43]    [38248998912.00]    [    4.11]
    [           24.43]    [17004999936.00]    [    7.57]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen. The first row contains column headers. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Retrieve Equity Screening Data Using Override Fields

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve equity screening data as of a specified date using these input arguments. The override field `PiTDate` is equivalent to the flag `AsOf` in the Bloomberg Excel Add-In.

- Bloomberg connection `c`
- Bloomberg screen is `Vehicle-Engine-Parts`
- Bloomberg screen type is `'GLOBAL'`
- Language is `'ENGLISH'`
- Bloomberg screen folder name is `'GENERAL'`
- Override field `PiTDate` is September 9, 2014

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL','ENGLISH','GENERAL',...
    'OverrideFields',{'PiTDate','20140909'});
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    'Ticker'        'Short Name'        'Market Cap'    'Price:D-1'       'P/E'
    'HON US'     'HONEYWELL INTL'      [7.3919e+10]    [  94.4600]    [17.8087]
    'TSLA US'    'TESLA MOTORS'        [3.4707e+10]    [ 278.4800]    [    NaN]

  Columns 6 through 8

    'Total Return YTD'    'Revenue T12M'    'EPS T12M'
    [          4.8907]    [  3.9966e+10]    [  5.1600]
    [         85.1239]    [  2.4365e+09]    [ -1.3500]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen as of September 9, 2014. The first row contains column headers. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

`sname` — Screen name

character vector | string scalar

Screen name, specified as a character vector or string scalar to denote the Bloomberg V3 session screen name to execute. The screen can be a customized equity screen or one of the Bloomberg example screens accessed by using the **EQS <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

`stype` — Screen type

`'GLOBAL'` | `'PRIVATE'`

Screen type, specified as one of the two preceding values to denote the Bloomberg screen type. `'GLOBAL'` denotes a Bloomberg screen name and `'PRIVATE'` denotes a customized screen name. When using the optional `group` input argument, `stype` cannot be set to `'PRIVATE'` for customized screen names.

`languageid` — Language identifier

character vector | string scalar

Language identifier, specified as a character vector or string to denote the language for the returned data. This argument is optional.

Data Types: `char` | `string`

`group` — Group identifier

character vector | string scalar

Group identifier, specified as a character vector or string to denote the Bloomberg screen folder name accessed by using the **EQS <GO>** option from the Bloomberg terminal. This argument is optional. When using this argument, `stype` cannot be set to `'PRIVATE'` for customized screen names.

Data Types: `char` | `string`

ov — Bloomberg override field values
cell array

Bloomberg override field values, specified as an n-by-2 cell array. The first column of the cell array is the override field. The second column is the override value.

Example: `{'PiTDate','20140909'}`

Data Types: `cell`

Output Arguments

d — Equity screening data
cell array (default) | structure | table

Equity screening data, returned as a cell array, structure, or table. The data type of the equity screening data depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `getdata` | `tahistory`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface” on page 5-28

fieldinfo

Field information for Bloomberg B-PIPE connection V3

Syntax

```
d = fieldinfo(c,f)
```

Description

`d = fieldinfo(c,f)` returns field information using the `bloombergBPIPE` object `c` with the Bloomberg B-PIPE C++ interface and field mnemonic `f`.

Examples

Retrieve Information for Last Price Field

Create a Bloomberg connection, and then retrieve information for the last price field.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return data as a table by setting the `DataReturnFormat` property of the `bloomberg` object. If you do not set this property, the `fieldinfo` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Retrieve the Bloomberg field information for the `LAST_PRICE` field.

```
f = 'LAST_PRICE';
d = fieldinfo(c,f);
```

Display the last four columns in the returned Bloomberg information.

```
d(:,2:5)
```

```
ans =

  1×4 table

     ID         MNEMONIC            DESCRIPTION          DATATYPE
    _____    _____________    ________________________    ________

    'RQ005'    'LAST_PRICE'    'Last Trade/Last Price'    'Double'
```

The columns in `d` are:

- Field identifier
- Field mnemonic
- Field name
- Field data type

You can also access the Bloomberg help information in the first column.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

f — Field mnemonic

character vector | string scalar

Field mnemonic, specified as a character vector or string scalar that denotes the Bloomberg field information to retrieve.

Data Types: `char` | `string`

Output Arguments

d — Field information

cell array (default) | structure | table

Field information, returned as an N-by-5 cell array, a structure, or a table.

The columns (or fields) of the data types are:

- Field help
- Field identifier
- Field mnemonic
- Field name

- Field data type

The data type of the field information depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `getdata` | `history` | `realtime` | `timeseries` | `category` | `fieldsearch`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface” on page 5-28
“Retrieve Bloomberg Historical Data Using Bloomberg B-PIPE C++ Interface” on page 5-31

fieldsearch

Field search for Bloomberg B-PIPE connection V3

Syntax

```
d = fieldsearch(c,f)
```

Description

`d = fieldsearch(c,f)` returns field information using the `bloombergBPIPE` object `c` with the Bloomberg B-PIPE C++ interface and search term `f`.

Examples

Retrieve Information for Last Price Field

Create a Bloomberg connection, and then retrieve information for the last price field.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return data as a table by setting the `DataReturnFormat` property of the `bloombergBPIPE` object. If you do not set this property, the `fieldsearch` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Return information for the search term `LAST_PRICE`.

```
f = 'LAST_PRICE';
d = fieldsearch(c,f);
```

Display the first three rows of the field information in `d`.

```
d(1:3,:)
```

```
ans =

  3×5 table

          CATEGORY             ID            MNEMONIC                 DESCRIPTION
    ___________________________    _______    ____________________    ___________________________________

    'Market Activity/Last'    'PR005'    'PX_LAST'                'Last Price'
    'Market Activity/Last'    'RQ005'    'LAST_PRICE'             'Last Trade/Last Price'
    'Market Activity/Last'    'PR910'    'CRNCY_ADJ_PX_LAST'      'Currency Adjusted Last Price'
```

The columns in `d` are:

- Category
- Field identifier
- Field mnemonic
- Field name
- Field data type

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

`f` — Search term

character vector | string scalar

Search term, specified as a character vector or string scalar that denotes the Bloomberg field descriptive data to retrieve.

Data Types: `char` | `string`

Output Arguments

`d` — Field data

cell array (default) | structure | table

Field data, returned as an N-by-5 cell array, a structure, or a table.

The columns (or fields) of the data types are:

- Category
- Field identifier
- Field mnemonic

- Field name
- Field data type

The data type of the field data depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `getdata` | `history` | `realtime` | `timeseries` | `category` | `fieldinfo`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface” on page 5-28
“Retrieve Bloomberg Historical Data Using Bloomberg B-PIPE C++ Interface” on page 5-31

get

Properties of Bloomberg B-PIPE connection V3

Syntax

```
v = get(c)
v = get(c,properties)
```

Description

`v = get(c)` returns a structure where each field name is the name of a property of the `bloombergBPIPE` object `c`, which uses the Bloomberg B-PIPE C++ interface, and each field contains the value of that property.

`v = get(c,properties)` returns the value of the specified properties `properties` for the Bloomberg V3 connection object.

Examples

Retrieve Bloomberg Connection Properties

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the Bloomberg connection properties.

```
v = get(c)
```

```
ans =

  struct with fields:

      session: [1×1 datafeed.internal.BLPSession]
    ipaddress: "localhost"
         port: 8194.00
```

v is a structure containing the Bloomberg session object, IP address, port number, and timeout value.

Close the Bloomberg connection.

```
close(c)
```

Retrieve One Bloomberg Connection Property

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

c is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the port number from the Bloomberg connection object by specifying `'port'` as a character vector.

```
property = "port";
v = get(c,property)
```

```
v =

        8194
```

v is a double that contains the port number of the Bloomberg connection object.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Two Bloomberg Connection Properties

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.

- The port number of the machine running the Bloomberg B-PIPE process is 8194.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Create a cell array `properties` with character vectors `'session'` and `'port'`. Retrieve the Bloomberg session object and port number from the Bloomberg connection object.

```
properties = ["session" "port"];
v = get(c,properties)
```

```
v =

  struct with fields:

    session: [1x1 com.bloomberglp.blpapi.Session]
       port: 8194
```

`v` is a structure containing the Bloomberg session object and port number.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

`properties` — Property names

character vector | string scalar | cell array of character vectors | string array

Property names, specified as a character vector, string scalar, cell array of character vectors, or string array containing Bloomberg connection property names. The property names are `session`, `ipaddress`, `port`, and `timeout`.

Data Types: `char` | `cell` | `string`

Output Arguments

`v` — Bloomberg connection properties

numeric scalar | character vector | object | structure

Bloomberg connection properties, returned as these data types depending on the requested properties.

Requested Properties	Data Type
Port number or timeout	Numeric scalar
IP address	Character vector
Bloomberg session	Object
All properties	Structure

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface” on page 5-28

getbulkdata

Bulk data with header information for Bloomberg B-PIPE connection V3

Syntax

```
d = getbulkdata(c,s,f)
d = getbulkdata(c,s,f,o,ov)
d = getbulkdata(c,s,f,o,ov,Name,Value)
[d,sec] = getbulkdata( ___ )
```

Description

`d = getbulkdata(c,s,f)` returns the bulk data for the fields `f` for the security list `s` using the `bloombergBPIPE` object `c` with the Bloomberg B-PIPE C++ interface.

`d = getbulkdata(c,s,f,o,ov)` returns the bulk data using the override fields `o` with corresponding override values `ov`.

`d = getbulkdata(c,s,f,o,ov,Name,Value)` returns the bulk data with additional options specified by one or more name-value pair arguments for Bloomberg request settings.

`[d,sec] = getbulkdata( ___ )` additionally returns the security list `sec` using any of the input argument combinations in the previous syntaxes.

Examples

Return Specific Field for Given Security

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return the dividend history for IBM.

```
security = 'IBM US Equity';
field = 'DVD_HIST'; % Dividend history field
```

```
[d,sec] = getbulkdata(c,security,field)
```

```
d =

    DVD_HIST: {{149x7 cell}}

sec =

    'IBM US Equity'
```

`d` is a structure with one field that contains a cell array with the returned bulk data. `sec` contains the IBM security name.

Display the dividend history with the associated header information by accessing the structure field `DVD_HIST`. This field is a cell array that contains one cell array. The nested cell array contains the dividend history data. Access the contents of the nested cell using cell array indexing.

```
d.DVD_HIST{1}
```

```
ans =

  Columns 1 through 6

    'Declared Date'    'Ex-Date'    'Record Date'    'Payable Date'    'Dividend Amount'    'Dividend Frequency'
    [       735536]    [ 735544]    [     735546]    [      735578]    [         0.95]      'Quarter'
    [       735445]    [ 735453]    [     735455]    [      735487]    [         0.95]      'Quarter'
    [       735354]    [ 735362]    [     735364]    [      735395]    [         0.95]      'Quarter'
    ...

  Column 7

    'Dividend Type'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
    ...
```

The first row of the dividend history data is the header information that describes the contents of each column.

Close the Bloomberg connection.

```
close(c)
```

Return Specific Field Using Override Values

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
```

```
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return the dividend history for IBM with dividend dates from January 1, 2004, through January 1, 2005.

```
security = 'IBM US Equity';
field = 'DVD_HIST';                          % Dividend history field
override = {'DVD_START_DT','DVD_END_DT'};  % Dividend start and
                                             % End dates
overridevalues = {'20040101','20050101'};

[d,sec] = getbulkdata(c,security,field,override,overridevalues)
```

```
d =

    DVD_HIST: {{5x7 cell}}

sec =

    'IBM US Equity'
```

`d` is a structure with one field that contains a cell array with the returned bulk data. `sec` contains the IBM security name.

Display the dividend history with the associated header information by accessing the structure field `DVD_HIST`. This field is a cell array that contains one cell array. The nested cell array contains the dividend history data. Access the contents of the nested cell using cell array indexing.

```
d.DVD_HIST{1}
```

```
ans =

  Columns 1 through 6

    'Declared Date'    'Ex-Date'    'Record Date'    'Payable Date'    'Dividend Amount'    'Dividend Frequency'
    [       732246]    [ 732259]    [     732261]    [      732291]    [         0.18]      'Quarter'
    [       732155]    [ 732165]    [     732169]    [      732200]    [         0.18]      'Quarter'
    [       732064]    [ 732073]    [     732077]    [      732108]    [         0.18]      'Quarter'
    [       731973]    [ 731983]    [     731987]    [      732016]    [         0.16]      'Quarter'

  Column 7

    'Dividend Type'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
```

The first row of the dividend history data is the header information that describes the contents of each column.

Close the Bloomberg connection.

```
close(c)
```

Return Specific Field Using Name-Value Pair Arguments

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return the closing price and dividend history for IBM with dividend dates from January 1, 2004, through January 1, 2005. Specify the data return format as a character vector by setting the name-value pair argument `'returnFormattedValue'` to `'true'`.

```
security = 'IBM US Equity';
fields = {'LAST_PRICE','DVD_HIST'};        % Closing price and
                                           % Dividend history fields
override = {'DVD_START_DT','DVD_END_DT'};  % Dividend start and
                                           % End dates
overridevalues = {'20040101','20050101'};

[d,sec] = getbulkdata(c,security,fields,override,overridevalues,...
                      'returnFormattedValue',true)
```

```
d =

      DVD_HIST: {{5x7 cell}}
    LAST_PRICE: {'188.74'}

sec =

    'IBM US Equity'
```

`d` is a structure with two fields. The first field `DVD_HIST` contains a cell array with the dividend historical data as a cell array. The second field `LAST_PRICE` contains a cell array with the closing price as a character vector. `sec` contains the IBM security name.

Display the closing price.

```
d.LAST_PRICE
```

```
ans =

    '188.74'
```

Display the dividend history with the associated header information by accessing the structure field `DVD_HIST`. This field is a cell array that contains one cell array. The nested cell array contains the dividend history data. Access the contents of the nested cell using cell array indexing.

```
d.DVD_HIST{1}
```

```
ans =

  Columns 1 through 6

    'Declared Date'    'Ex-Date'    'Record Date'    'Payable Date'    'Dividend Amount'    'Dividend Frequency'
    [       732246]    [ 732259]    [     732261]    [      732291]    [         0.18]      'Quarter'
    [       732155]    [ 732165]    [     732169]    [      732200]    [         0.18]      'Quarter'
    [       732064]    [ 732073]    [     732077]    [      732108]    [         0.18]      'Quarter'
    [       731973]    [ 731983]    [     731987]    [      732016]    [         0.16]      'Quarter'

  Column 7

    'Dividend Type'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
```

The first row of the dividend history data is the header information that describes the contents of each column.

Close the Bloomberg connection.

```
close(c)
```

Return Bulk Data as Table with Datetime

Create a Bloomberg connection, and then request dividend history data. The `getbulkdata` function returns data for dates as a `datetime` array.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `getbulkdata` function returns data as a structure.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Return the dividend history for IBM.

```
s = 'IBM US Equity';
f = 'DVD_HIST'; % Dividend history field

d = getbulkdata(c,s,f);
```

Display the first three rows of the table.

```
d.DVD_HIST{1}(1:3,:)
```

```
ans =

  3×7 table

       DeclaredDate            ExmDate             RecordDate            PayableDate
    ____________________    ____________________    ____________________    ____________________

    31-Oct-2017 00:00:00    09-Nov-2017 00:00:00    10-Nov-2017 00:00:00    09-Dec-2017 00:00:00
    25-Jul-2017 00:00:00    08-Aug-2017 00:00:00    10-Aug-2017 00:00:00    09-Sep-2017 00:00:00
    25-Apr-2017 00:00:00    08-May-2017 00:00:00    10-May-2017 00:00:00    10-Jun-2017 00:00:00
```

Display three declared dates. The `DeclaredDate` variable is a `datetime` array.

```
d.DVD_HIST{1}.DeclaredDate(1:3)
```

```
ans =

  3×1 datetime array

   31-Oct-2017 00:00:00
   25-Jul-2017 00:00:00
   25-Apr-2017 00:00:00
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

`f` — Bloomberg data fields

character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

`o` — Bloomberg override field

`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field name. A cell array of character vectors or string array denotes multiple Bloomberg override field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `'END_DT'`

Data Types: `char` | `cell` | `string`

`ov` — Bloomberg override field value

`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field value, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field value. A cell array of character vectors or string array denotes multiple Bloomberg override field values. Use this field value to filter the Bloomberg data result set.

Example: `'20100101'`

Data Types: `char` | `cell` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'returnFormattedValue',true`

`returnEids` — Entitlement identifiers

true | false

Entitlement identifiers, specified as the comma-separated pair consisting of `'returnEids'` and a Boolean. `true` adds a name and value for the entitlement identifier (EID) date to the return data.

Data Types: `logical`

`returnFormattedValue` — Return format
true | false

Return format, specified as the comma-separated pair consisting of `'returnFormattedValue'` and a Boolean. `true` forces all data to be returned as the data type character vector.

Data Types: `logical`

`useUTCTime` — Date time format
true | false

Date time format, specified as the comma-separated pair consisting of `'useUTCTime'` and a Boolean. `true` returns date and time values as Coordinated Universal Time (UTC) and `false` defaults to the Bloomberg **TZDF <GO>** settings of the requestor.

Data Types: `logical`

`forcedDelay` — Latest reference data
true | false

Latest reference data, specified as the comma-separated pair consisting of `'forcedDelay'` and a Boolean. `true` returns the latest data up to the delay period specified by the exchange for the security.

Data Types: `logical`

Output Arguments

`d` — Bloomberg data
structure (default) | table | timetable

Bloomberg data, returned as a structure, table, or timetable. The data type of the Bloomberg data depends on the DataReturnFormat and DatetimeType properties of the connection object. For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`sec` — Security list
cell array of character vectors

Security list, returned as a cell array of character vectors for the corresponding securities in `s`. The contents of `sec` are identical in value and order to `s`. You can return securities with any of the following identifiers:

- `buid`
- `cats`
- `cins`
- `common`
- `cusip`
- `isin`
- `sedol1`
- `sedol2`
- `sicovam`

- `svm`
- `ticker` (default)
- `wpk`

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface” on page 5-28

getdata

Current data for Bloomberg B-PIPE connection V3

Syntax

```
d = getdata(c,s,f)
d = getdata(c,s,f,o,ov)
d = getdata(c,s,f,o,ov,Name,Value)
[d,sec] = getdata( ___ )
```

Description

`d = getdata(c,s,f)` returns the data for the fields `f` for the security list `s` using the `bloombergBPIPE` object with the Bloomberg B-PIPE C++ interface. `getdata` accesses the Bloomberg reference data service.

`d = getdata(c,s,f,o,ov)` returns the data using the override fields `o` with corresponding override values `ov`.

`d = getdata(c,s,f,o,ov,Name,Value)` returns the data using name-value pair arguments for additional Bloomberg request settings.

`[d,sec] = getdata( ___ )` additionally returns the security list `sec` using any of the input argument combinations in the previous syntaxes.

Examples

Last and Open Price for Security

First, create a Bloomberg B-PIPE connection. Then, request last and open prices for a security. The current data you see when running this code can differ from the output data here.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Request last and open prices for Microsoft.

```
[d,sec] = getdata(c,'MSFT US Equity',{'LAST_PRICE';'OPEN'})
```

```
d =
    LAST_PRICE: 33.3401
          OPEN: 33.6000

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the security in `sec`.

Close the connection.

```
close(c)
```

Specified Fields Given Override Fields and Values

First, create a Bloomberg B-PIPE connection. Then, request data for specific fields for a security using an override field and value. The current data you see when running this code can differ from the output data here.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Request data for Bloomberg fields `'YLD_YTM_ASK'`, `'ASK'`, and `'OAS_SPREAD_ASK'` when the Bloomberg field `'OAS_VOL_ASK'` is `'14.000000'`.

```
[d,sec] = getdata(c,'030096AF8 Corp',...
   {'YLD_YTM_ASK','ASK','OAS_SPREAD_ASK','OAS_VOL_ASK'},...
   {'OAS_VOL_ASK'},{'14.000000'})
```

```
d =
       YLD_YTM_ASK: 5.6763
               ASK: 120.7500
    OAS_SPREAD_ASK: 307.9824
       OAS_VOL_ASK: 14
```

```
sec =
    '030096AF8 Corp'
```

`getdata` returns a structure `d` with the resulting values for the requested fields.

Close the connection.

```
close(c)
```

Request for Security Using CUSIP Number

First, create a Bloomberg B-PIPE connection. Then, use the CUSIP number for a security to request last price. The current data you see when running this code can differ from the output data here.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Request the last price for IBM with the CUSIP number.

```
d = getdata(c,'/cusip/459200101','LAST_PRICE')
```

```
d =
    LAST_PRICE: 182.5100
```

`getdata` returns a structure `d` with the last price.

Close the connection.

```
close(c)
```

Last Price for Security with Pricing Source

First, create a Bloomberg B-PIPE connection. Then, request the last price for a security. Specify the security using the CUSIP number with a pricing source. The current data you see when running this code can differ from the output data here.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Specify IBM with the CUSIP number and the pricing source `BGN` after the `@` symbol.

```
d = getdata(c,'/cusip/459200101@BGN','LAST_PRICE')
```

```
d =
    LAST_PRICE: 186.81
```

`getdata` returns a structure `d` with the last price.

Close the connection.

```
close(c)
```

Constituent Weights Using Date Override

First, create a Bloomberg B-PIPE connection. Then, request the constituent weights of an index using a date override. The current data you see when running this code can differ from the output data here.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return the constituent weights for the Dow Jones Index as of January 1, 2010, using a date override with the required date format `YYYYMMDD`.

```
d = getdata(c,'DJX Index','INDX_MWEIGHT','END_DT','20100101')

d =
    INDX_MWEIGHT: {{30x2 cell}}
```

`getdata` returns a structure `d` with a cell array where the first column is the index and the second column is the constituent weight.

Display the constituent weights for each index.

```
d.INDX_MWEIGHT{1,1}

ans =
    'AA UN'      [1.1683]
    'AXP UN'     [2.9366]
    'BA UN'      [3.9229]
    'BAC UN'     [1.0914]
    ...
```

Close the connection.

```
close(c)
```

Current Data and Dates as Table with Datetime

Create a Bloomberg connection, and then request current data for specific fields. The `getdata` function returns data for dates as a `datetime` array.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `getdata` function returns data as a structure.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Request current data for these fields:

- Last update date
- Last price
- Number of trades
- Previous real-time trading date

```
s = 'IBM US Equity';
f = {'LAST_UPDATE_DT','LAST_PRICE', ...
    'NUM_TRADES_RT','PREV_TRADING_DT_REALTIME'};
d = getdata(c,s,f)
```

```
d =

  1×4 table

      LAST_UPDATE_DT       LAST_PRICE    NUM_TRADES_RT    PREV_TRADING_DT_REALTIME
    ___________________    __________    _____________    ________________________

    21-Dec-2017 00:00:00      152.2           24846          20-Dec-2017 00:00:00
```

Display the last update date. This date is a `datetime` array.

```
d.LAST_UPDATE_DT
```

```
ans =

  datetime

   21-Dec-2017 00:00:00
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

f — Bloomberg data fields

character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

o — Bloomberg override field
`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field name. A cell array of character vectors or string array denotes multiple Bloomberg override field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `'END_DT'`

Data Types: `char` | `cell` | `string`

ov — Bloomberg override field value
`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field value, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field value. A cell array of character vectors or string array denotes multiple Bloomberg override field values. Use this field value to filter the Bloomberg data result set.

Example: `'20100101'`

Data Types: `char` | `cell` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'returnEids',true`

`returnEids` — Entitlement identifiers
true | false

Entitlement identifiers, specified as the comma-separated pair consisting of `'returnEids'` and a Boolean. `true` adds a name and value for the entitlement identifier (EID) date to the return data.

Data Types: `logical`

`returnFormattedValue` — Return format
true | false

Return format, specified as the comma-separated pair consisting of `'returnFormattedValue'` and a Boolean. `true` forces all data to be returned as the data type character vector.

Data Types: `logical`

`useUTCTime` — Date time format
true | false

Date time format, specified as the comma-separated pair consisting of `'useUTCTime'` and a Boolean. `true` returns date and time values as Coordinated Universal Time (UTC) and `false` defaults to the Bloomberg **TZDF <GO>** settings of the requestor.

Data Types: `logical`

`forcedDelay` — Latest reference data
true | false

Latest reference data, specified as the comma-separated pair consisting of `'forcedDelay'` and a Boolean. `true` returns the latest data up to the delay period specified by the exchange for the security.

Data Types: `logical`

Output Arguments

`d` — Bloomberg data
structure (default) | table | timetable

Bloomberg data, returned as a structure, table, or timetable. The data type of the Bloomberg data depends on the DataReturnFormat and DatetimeType properties of the connection object. For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`d` returns an additional field named `EID`, which means entitlement identifier. For details, see the *Bloomberg API Developer's Guide*.

`sec` — Security list
cell array of character vectors

Security list, returned as a cell array of character vectors for the corresponding securities in `s`. The contents of `sec` are identical in value and order to `s`. You can return securities with any of the following identifiers:

- `buid`
- `cats`
- `cins`
- `common`
- `cusip`
- `isin`
- `sedol1`
- `sedol2`
- `sicovam`
- `svm`
- `ticker` (default)

- `wpk`

Tips

- Bloomberg V3 data supports additional name-value pair arguments. To access further information on these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.
- You can check data and field availability by using the Bloomberg Excel Add-In.

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `history` | `realtime` | `timeseries`

Topics

"Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface" on page 5-28

history

Historical data for Bloomberg B-PIPE connection V3

Syntax

```
d = history(c,s,f,fromdate,todate)
d = history(c,s,f,fromdate,todate,period)
d = history(c,s,f,fromdate,todate,period,currency)
d = history(c,s,f,fromdate,todate,period,currency,Name,Value)
[d,sec] = history( ___ )
```

Description

`d = history(c,s,f,fromdate,todate)` returns the historical data for the security list `s` for the fields `f` for the dates `fromdate` through `todate` using the `bloombergBPIPE` object `c` with the Bloomberg B-PIPE C++ interface. Date strings can be input in any format recognized by MATLAB. `sec` is the security list that maps the order of the return data. The return data `d` is sorted to match the input order of `s`.

`d = history(c,s,f,fromdate,todate,period)` returns the historical data for the fields `f` and the dates `fromdate` through `todate` with a specific periodicity `period`.

`d = history(c,s,f,fromdate,todate,period,currency)` returns the historical data for the security list `s` for the fields `f` and the dates `fromdate` through `todate` based on the given currency `currency`.

`d = history(c,s,f,fromdate,todate,period,currency,Name,Value)` returns the historical data for the security list `s` using additional options specified by one or more name-value pair arguments.

`[d,sec] = history( ___ )` additionally returns the security list `sec` using any of the input argument combinations in the previous syntaxes. The return data, `d` and `sec`, are sorted to match the input order of `s`.

Examples

Daily Closing Prices Within Date Range

First, create a Bloomberg Desktop connection. Then, retrieve the daily closing price for a security within a date range.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.

- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Get the daily closing price from August 1, 2010, through August 10, 2010, for the IBM security.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','8/10/2010')
```

```
d =

     734352.00        123.55
     734353.00        123.18
     734354.00        124.03
     734355.00        124.56
     734356.00        123.58
     734359.00        125.34
     734360.00        125.19


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Monthly Closing Prices Within Date Range

First, create a Bloomberg Desktop connection. Then, retrieve the monthly closing prices for a security within a date range.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Get the monthly closing price from August 1, 2010, through December 10, 2010, for the IBM security.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','12/10/2010','monthly')

d =

     734360.00        125.19
     734391.00        121.53
     734421.00        131.85
     734452.00        139.78
     734482.00        138.13


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Monthly Closing Prices Within Date Range Using US Currency

First, create a Bloomberg Desktop connection. Then, retrieve the monthly closing prices for a security within a date range. Specify prices using the US currency.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Get the monthly closing price from August 1, 2010, through December 10, 2010, for the IBM security in US currency `'USD'`.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','12/10/2010','monthly','USD')
```

```
d =

     734360.00        125.19
     734391.00        121.53
     734421.00        131.85
     734452.00        139.78
     734482.00        138.13


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Monthly Closing Prices Within Date Range Using Currency with Specified Period

First, create a Bloomberg Desktop connection. Then, retrieve the monthly closing prices for a security within a date range. Specify prices using the US currency. Specify period values to customize the returned data.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Get the monthly closing price from August 1, 2010, through August 1, 2011, for the IBM security in US currency. The period values `'monthly'`, `'actual'`, and `'all_calendar_days'` specify returning actual monthly data for all calendar days. The period value `'nil_value'` specifies filling missing data values with a `NaN`.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','8/01/2011',{'monthly','actual',...
                  'all_calendar_days','nil_value'},'USD')
```

```
d =

     734351.00        128.40
     734382.00        125.77
     734412.00        135.64
     734443.00        143.32
     734473.00        144.41
     734504.00        146.76
     734535.00        163.56
     734563.00        159.97
     734594.00        164.27
     734624.00        170.58
     734655.00        166.56
     734685.00        174.54
     734716.00        180.75


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Daily Closing Prices Within Date Range Using Currency with Name-Value Pairs

First, create a Bloomberg Desktop connection. Then, retrieve the daily closing prices for a security within a date range. Specify prices using the US currency. Use name-value pair arguments to adjust the prices.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Get the daily closing price from August 1, 2010, through August 10, 2010, for the IBM security in U.S. currency `'USD'`. The prices are adjusted for normal cash and splits.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','8/10/2010','daily','USD',...
                  'adjustmentNormal',true,...
                  'adjustmentSplit',true)
```

```
d =

     734352.00        123.55
     734353.00        123.18
     734354.00        124.03
     734355.00        124.56
     734356.00        123.58
     734359.00        125.34
     734360.00        125.19


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Daily Closing Prices Using CUSIP Number and Pricing Source

First, create a Bloomberg Desktop connection. Then, retrieve the daily closing prices for a security within a date range. Specify the security using the CUSIP number and a pricing source.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Get the daily closing price from January 1, 2012, through January 1, 2013, for the security specified with a CUSIP number `/cusip/459200101` and with pricing source `BGN`.

`d` contains the numeric representation for the date in the first column and the closing price in the second column.

```
d = history(c,'/cusip/459200101@BGN','LAST_PRICE',...
            '01/01/2012','01/01/2013')
```

```
d =

     734871.00        180.69
     734872.00        179.96
     734873.00        179.10
     ...
```

Close the Bloomberg connection.

```
close(c)
```

Closing Prices Within Date Range Using International Date Format

First, create a Bloomberg Desktop connection. Then, retrieve the closing prices for a security within a date range. Specify the dates for the range using an international date format.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return the closing price for the given dates in international format for the security `'MSFT@BGN US Equity'`.

```
stDt = datenum('01/06/11','dd/mm/yyyy');
endDt = datenum('01/06/12','dd/mm/yyyy');
[d,sec] = history(c,'MSFT@BGN US Equity','LAST_PRICE',...
                  stDt,endDt,{'previous_value','all_calendar_days'})
```

```
d =

     734655.00         22.92
     734656.00         22.72
     734657.00         22.42
     ...
```

```
sec =

    'MSFT@BGN US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Median Estimated Earnings Per Share Using Override Fields

First, create a Bloomberg Desktop connection. Then, retrieve the median earnings per share for a security within a date range. Specify an override field and value.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the median estimated earnings per share for AkzoNobel from October 1, 2010, through October 30, 2010. When specifying Bloomberg override fields, use the character vector `'overrideFields'`. The `overrideFields` argument must be an `n`-by-`2` cell array, where the first column is the override field and the second column is the override value.

```
d = history(c,'AKZA NA Equity', ...
            'BEST_EPS_MEDIAN',datenum('01.10.2010', ...
            'dd.mm.yyyy'),datenum('30.10.2010','dd.mm.yyyy'), ...
            {'daily','calendar'},[],'overrideFields', ...
            {'BEST_FPERIOD_OVERRIDE','BF'})

d =

     734412.00          3.75
     734415.00          3.75
     734416.00          3.75
     ...
```

`d` returns the numeric representation for the date in the first column and the median estimated earnings per share in the second column.

Close the Bloomberg connection.

```
close(c)
```

Historical Data as Table with Dates

Create a Bloomberg connection, and then retrieve closing prices for a historical date range. The `history` function returns data for dates as a `datetime` array.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `history` function returns data as a numeric array.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve historical closing prices for IBM from August 1, 2010, through August 10, 2010. `d` is a table that contains dates as a `datetime` array.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE', ...
    '8/01/2010','8/10/2010')


d =

  7×2 table

       DATE        LAST_PRICE
    ___________    __________

    02-Aug-2010      130.76
```

```
    03-Aug-2010      130.37
    04-Aug-2010      131.27
    05-Aug-2010      131.83
    06-Aug-2010      130.14
    09-Aug-2010      132.00
    10-Aug-2010      131.84


sec =

  1×1 cell array

    {'IBM US Equity'}
```

Access dates in the returned data.

```
d.DATE
```

```
ans =

  7×1 datetime array

   02-Aug-2010
   03-Aug-2010
   04-Aug-2010
   05-Aug-2010
   06-Aug-2010
   09-Aug-2010
   10-Aug-2010
```

Close the Bloomberg connection.

```
close(c)
```

Historical Data as Timetable

Create a Bloomberg connection, and then retrieve closing prices for a historical date range. The `history` function returns data as a `timetable`.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
```

```
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return data as a `timetable` by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `history` function returns data as a numeric array.

```
c.DataReturnFormat = 'timetable';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve historical closing prices for IBM from August 1, 2010, through August 10, 2010. `d` is a `timetable` that contains dates in the first column.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE', ...
    '8/01/2010','8/10/2010')
```

```
d =

  7×1 timetable

       DATE        LAST_PRICE
    ____________    __________

    02-Aug-2010       130.76
    03-Aug-2010       130.37
    04-Aug-2010       131.27
    05-Aug-2010       131.83
    06-Aug-2010       130.14
    09-Aug-2010       132.00
    10-Aug-2010       131.84


sec =

  1×1 cell array

    {'IBM US Equity'}
```

Access dates in the returned data.

```
d.DATE
```

```
ans =

  7×1 datetime array

   02-Aug-2010
   03-Aug-2010
   04-Aug-2010
   05-Aug-2010
   06-Aug-2010
   09-Aug-2010
```

```
10-Aug-2010
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

f — Bloomberg data fields

character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

period — Periodicity

`'daily'` | `'weekly'` | `'monthly'` | `'quarterly'` | ...

Periodicity, specified as one of these values to denote the data to return. For specifying multiple values, use a cell array. For example, when `period` is set to `{'daily','all_calendar_days'}`, `history` returns daily data for all calendar days, and reports missing data as `NaN`s. When `period` is set to `'active_days_only'`, `history` returns data using the default periodicity for active trading days only. The default periodicity depends on the security. If a security is reported on a monthly basis, the default periodicity is monthly. These tables show the values for `period`.

To specify the periodicity of the return data, see this table.

Value	Description
`'daily'`	Return data for each day.
`'weekly'`	Return data for each week.
`'monthly'`	Return data for each month.
`'quarterly'`	Return data for each quarter.

Value	Description
`'semi_annually'`	Return data semiannually.
`'yearly'`	Return data for each year.

The anchor date is the date to which all other reported dates are related. To specify the anchor date, see this table.

Value	Description
`'actual'`	Anchor date specification for an actual date. For this function, for periodicities other than daily, `todate` is the anchor date. If the period is weekly and the `todate` is a Thursday, every data point is a Thursday, or the nearest prior business day to Thursday. If the period is monthly and the `todate` is the 20th of a month, every data point is the 20th of each month in the date range.
`'calendar'`	Anchor date specification for a calendar year.
`'fiscal'`	Anchor date specification for a fiscal year.
`'none'`	Do not specify the anchor date.

To specify returning data for particular days, see this table.

Value	Description
`'non_trading_weekdays'`	Return data for all weekdays.
`'all_calendar_days'`	Return data for all calendar days.
`'active_days_only'`	Return data for only active trading days.

To specify how to fill missing values, see this table.

Value	Description
`'previous_value'`	Fill missing values with previous values for dates without trading activity for the security. If no previous value exists in the month before the `fromdate`, this function retains the missing values.
`'nil_value'`	Fill missing values with a `NaN` for dates without trading activity for the security.

Data Types: `char` | `cell`

`currency` — Currency
character vector | string scalar

Currency, specified as a character vector or string scalar to denote the ISO code for the currency of the returned data. For example, to specify output money values in U.S. currency, use `USD` for this argument.

Data Types: `char` | `string`

`fromdate` — Beginning date
double scalar | character vector | string scalar | datetime

Beginning date for the historical data, specified as a double scalar, character vector, string scalar, or `datetime` array. You can specify dates in any of the formats supported by `datestr` and `datenum` that show a year, month, and day.

Data Types: `datetime` | `double` | `char` | `string`

`todate` — End date
double scalar | character vector | string scalar | datetime

End date for the historical data, specified as a double scalar, character vector, string scalar, or `datetime` array. You can specify dates in any of the formats supported by `datestr` and `datenum` that show a year, month, and day.

Data Types: `datetime` | `double` | `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'adjustmentNormal',true`

`overrideFields` — Override fields
cell array

Override fields, specified as the comma-separated pair consisting of `'overrideFields'` and an `n`-by-`2` cell array. The first column of the cell array is the override field and the second column is the override value.

Example: `'overrideFields',
{'IVOL_DELTA_LEVEL','DELTA_LVL_10';'IVOL_DELTA_PUT_OR_CALL','IVOL_PUT';'IVOL_MATURITY','MATURITY_1STM'}`

Data Types: `cell`

`adjustmentNormal` — Historical normal pricing adjustment
true | false

Historical normal pricing adjustment, specified as the comma-separated pair consisting of `'adjustmentNormal'` and a Boolean to reflect:

- Regular Cash
- Interim
- 1st Interim
- 2nd Interim
- 3rd Interim
- 4th Interim

- 5th Interim
- Income
- Estimated
- Partnership Distribution
- Final
- Interest on Capital
- Distribution
- Prorated

For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

`adjustmentAbnormal` — Historical abnormal pricing adjustment
true | false

Historical abnormal pricing adjustment, specified as the comma-separated pair consisting of `'adjustmentAbnormal'` and a Boolean to reflect:

- Special Cash
- Liquidation
- Capital Gains
- Long-Term Capital Gains
- Short-Term Capital Gains
- Memorial
- Return of Capital
- Rights Redemption
- Miscellaneous
- Return Premium
- Preferred Rights Redemption
- Proceeds/Rights
- Proceeds/Shares
- Proceeds/Warrants

For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

`adjustmentSplit` — Historical split pricing or volume adjustment
true | false

Historical split pricing or volume adjustment, specified as the comma-separated pair consisting of `'adjustmentSplit'` and a Boolean to reflect:

- Spin-Offs

- Stock Splits/Consolidations
- Stock Dividend/Bonus
- Rights Offerings/Entitlement

For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

`adjustmentFollowDPDF` — Historical pricing adjustment
true (default) | false

Historical pricing adjustment, specified as the comma-separated pair consisting of `'adjustmentFollowDPDF'` and a Boolean. Setting this name-value pair follows the **DPDF <GO>** option from the Bloomberg terminal. For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

Output Arguments

`d` — Bloomberg historical data
numeric array (default) | table | timetable

Bloomberg historical data, returned as a numeric array, table, or timetable. The data type of the historical data depends on the DataReturnFormat and DatetimeType properties of the connection object. The first column (or field) in the historical data contains the date. The remaining columns contain the requested data fields.

For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`sec` — Security list
cell array of character vectors

Security list, returned as a cell array of character vectors for the corresponding securities in `s`. The contents of `sec` are identical in value and order to `s`. You can return securities with any of the following identifiers:

- `buid`
- `cats`
- `cins`
- `common`
- `cusip`
- `isin`
- `sedol1`
- `sedol2`
- `sicovam`
- `svm`
- `ticker` (default)

- `wpk`

Tips

- You can check data and field availability by using the Bloomberg Excel Add-In.

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `getdata` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Historical Data Using Bloomberg B-PIPE C++ Interface” on page 5-31

isconnection

Determine Bloomberg B-PIPE connection V3

Syntax

```
v = isconnection(c)
```

Description

`v = isconnection(c)` returns `true` (`1`) if `c` is a valid Bloomberg V3 connection using the Bloomberg B-PIPE C++ interface and `false` (`0`) otherwise.

Examples

Validate the Bloomberg Connection

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface” on page 5-28

lookup

Find information about securities for Bloomberg B-PIPE connection V3

Syntax

```
l = lookup(c,q,reqtype,Name,Value)
```

Description

`l = lookup(c,q,reqtype,Name,Value)` retrieves data based on criteria in the query `q` for a specific request type `reqtype` using the Bloomberg connection `c` with the Bloomberg B-PIPE C++ interface. For additional information about the query criteria and the possible name-value pair combinations, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Examples

Look Up Security

Create a Bloomberg connection, and then use the Security Lookup to retrieve information about the IBM corporate bond. For details about Bloomberg and the parameter values you can set, see the *Bloomberg API Developer's Guide* using the **WAPI<GO>** option from the Bloomberg terminal.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `lookup` function returns data as a structure.

```
c.DataReturnFormat = 'table';
```

Retrieve the instrument data for an IBM corporate bond with a maximum of 20 rows of data. The Security Lookup returns the security names and descriptions.

```
insts = lookup(c,'IBM','instrumentListRequest','maxResults',20, ...
    'yellowKeyFilter','YK_FILTER_CORP', ...
    'languageOverride','LANG_OVERRIDE_NONE');
```

Display the first three rows in the table. The first column contains the IBM corporate bond names, and the second column contains the bond descriptions.

```
insts(1:3,:)
```

```
ans =

  3×2 table

       security                    description
    ________________    ____________________________________

    'DD103619 <corp>'    'International Business Machines Corp'
    '459200AG <corp>'    'International Business Machines Corp'
    'EC767659 <corp>'    'International Business Machines Corp'
```

Close the Bloomberg connection.

```
close(c)
```

Look Up Curve

Use the Curve Lookup to retrieve information about the `'GOLD'` related curve `'CD1016'`. For details about Bloomberg and the parameter values you can set, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the curve data for the credit default swap subtype of corporate bonds for a `'GOLD'` related curve `'CD1016'`. Return a maximum of `10` rows of data for the U.S. with `'USD'` currency.

```
curves = lookup(c,'GOLD','curveListRequest','maxResults',10,...
                'countryCode','US','currencyCode','USD',...
                'curveid','CD1016','type','CORP','subtype','CDS')

curves =

            curve: {'YCCD1016 Index'}
      description: {'Goldman Sachs Group Inc/The'}
          country: {'US'}
         currency: {'USD'}
          curveid: {'CD1016'}
             type: {'CORP'}
          subtype: {'CDS'}
        publisher: {'Bloomberg'}
            bbgid: {''}
```

One row of data displays as Bloomberg curve name `'YCCD1016 Index'` with Bloomberg description `'Goldman Sachs Group Inc/The'` in the U.S. with `'USD'` currency. The Bloomberg short-form identifier for the curve is `'CD1016'`. Bloomberg is the publisher and the `bbgid` is blank.

Close the Bloomberg connection.

```
close(c)
```

Look Up Government Security

Use the Government Security Lookup to retrieve information for United States Treasury bonds. For details about Bloomberg and the parameter values you can set, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Filter government security data with ticker filter of `'T'` for a maximum of `10` rows of data.

```
govts = lookup(c,'T','govtListRequest','maxResults',10,...
               'partialMatch',false)

govts =
```

```
      parseky: {10x1 cell}
         name: {10x1 cell}
       ticker: {10x1 cell}
```

The Government Security Lookup returns `parseky` data, the name, and ticker of the United States Treasury bonds.

Display the `parseky` data.

```
govts.parseky
```

```
ans =
    '912828VS Govt'
    '912828RE Govt'
    '912810RC Govt'
    '912810RB Govt'
    '912828VU Govt'
    '912828VV Govt'
    '912828VB Govt'
    '912828VR Govt'
    '912828VW Govt'
    '912828VQ Govt'
```

Display the names of the United States Treasury bonds.

```
govts.name
```

```
ans =
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
```

Display the tickers of the United States Treasury bonds.

```
govts.ticker
```

```
ans =
    'T'
    'T'
    'T'
    'T'
    'T'
    'T'
    'T'
    'T'
    'T'
    'T'
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

q — Keyword query

character vector | string scalar | cell array of character vectors | string array

Keyword query, specified as a character vector, string scalar, cell array of character vectors, or string array. Each character vector or string denotes an item for which information is requested. For example, the keyword query can be a security, a curve type, or a filter ticker.

Data Types: `char` | `cell` | `string`

reqtype — Request type

`'instrumentListRequest'` | `'curveListRequest'` | `'govtListRequest'`

Request type, specified as the preceding values to denote the type of information request. `'instrumentListRequest'` denotes a security or instrument lookup request. `'curveListRequest'` denotes a curve lookup request. `'govtListRequest'` denotes a government lookup request for government securities.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'maxResults'`, `20`, `'yellowKeyFilter'`, `'YK_FILTER_CORP'`, `'languageOverride'`, `'LANG_OVERRIDE_NONE'`, `'countryCode'`, `'US'`, `'currencyCode'`, `'USD'`, `'curveid'`, `'CD1016'`, `'type'`, `'CORP'`, `'subtype'`, `'CDS'`, `'partialMatch'`, `false`

maxResults — Number of rows in result data

numeric scalar

Number of rows in the result data, specified as the comma-separated pair consisting of `'maxResults'` and a numeric scalar to denote the total maximum number of rows of information to return. Result data can be one or more rows of data no greater than the number specified.

Data Types: `double`

yellowKeyFilter — Bloomberg yellow key filter

character vector | string scalar

Bloomberg yellow key filter, specified as the comma-separated pair consisting of `'yellowKeyFilter'` and a unique character vector or string scalar to denote the particular yellow key for government securities, corporate bonds, equities, and commodities, for example.

Data Types: `char` | `string`

`languageOverride` — Language override
character vector | string scalar

Language override, specified as the comma-separated pair consisting of `'languageOverride'` and a unique character vector or string scalar to denote a translation language for the result data.

Data Types: `char` | `string`

`countryCode` — Country code
character vector | string scalar

Country code, specified as the comma-separated pair consisting of `'countryCode'` and a character vector or string scalar to denote the country for the result data.

Data Types: `char` | `string`

`currencyCode` — Currency code
character vector | string scalar

Currency code, specified as the comma-separated pair consisting of `'currencyCode'` and a character vector or string scalar to denote the currency for the result data.

Data Types: `char` | `string`

`curveID` — Bloomberg short-form identifier for curve
character vector | string scalar

Bloomberg short-form identifier for a curve, specified as the comma-separated pair consisting of `'curveID'` and a character vector or string scalar.

Data Types: `char` | `string`

`type` — Bloomberg market sector type
character vector | string scalar

Bloomberg market sector type corresponding to the Bloomberg yellow keys, specified as the comma-separated pair consisting of `'type'` and a character vector or string scalar.

Data Types: `char` | `string`

`subtype` — Bloomberg market sector subtype
character vector | string scalar

Bloomberg market sector subtype, specified as the comma-separated pair consisting of `'subtype'` and a character vector or string scalar to further delineate the market sector type.

Data Types: `char` | `string`

`partialMatch` — Partial match on ticker
`true` | `false`

Partial match on ticker, specified as the comma-separated pair consisting of `'partialMatch'` and `true` or `false`. When set to `true`, you can filter securities by setting `q` to a query such as `'T*'`. When set to `false`, the securities are unfiltered.

Data Types: `logical`

Output Arguments

`l` — Lookup information

structure (default) | table

Lookup information, returned as a structure or table containing set properties depending on the request type. The data type of the lookup information depends on the DataReturnFormat property of the connection object.

For a list of the set properties and their descriptions, see the following tables.

'instrumentListRequest' Properties

Property	Description
`security`	Security name
`description`	Security long name

'curveListRequest' Properties

Property	Description
`curve`	Bloomberg curve name
`description`	Bloomberg description
`country`	Country code
`currency`	Currency code
`curveid`	Bloomberg short-form identifier for the curve
`type`	Bloomberg market sector type
`subtype`	Bloomberg market sector subtype
`publisher`	Bloomberg specified as publisher
`bbgid`	Bloomberg identifier

'govtListRequest' Properties

Property	Description
`parsekey`	Bloomberg security identifier (ticker or CUSIP, for example), price source, and source key (Bloomberg yellow key)
`name`	Government security name
`ticker`	Government security ticker

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface” on page 5-28

portfolio

Current portfolio data for Bloomberg B-PIPE connection V3

Syntax

```
d = portfolio(c,p,f)
d = portfolio(c,p,f,o,ov)
[d,plist] = portfolio( ___ )
```

Description

`d = portfolio(c,p,f)` returns current portfolio data for the fields `f` in the portfolio `p` using the `bloombergBPIPE` object `c`.

`d = portfolio(c,p,f,o,ov)` returns current portfolio data using override field `o` and override value `ov`.

`[d,plist] = portfolio( ___ )` also returns the portfolio list `plist` using any of the input argument combinations in the previous syntaxes.

Examples

Request Portfolio Data

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Request portfolio data for a custom portfolio with portfolio identifier `U335877-1 Client`. Request data using all fields `f`.

```
p = 'U335877-1 Client';
f = {'PORTFOLIO_MEMBERS','PORTFOLIO_MPOSITION',...
     'PORTFOLIO_MWEIGHT','PORTFOLIO_DATA'};
```

```
d = portfolio(c,p,f)
```

```
d =

    PORTFOLIO_MPOSITION: {{0x1 cell}}
      PORTFOLIO_MWEIGHT: {{0x1 cell}}
         PORTFOLIO_DATA: {{0x1 cell}}
      PORTFOLIO_MEMBERS: {{0x1 cell}}
```

`d` is a structure that contains portfolio data. Each structure field corresponds to data for each portfolio field.

Close the connection.

```
close(c)
```

Request Portfolio Data Using Specific Date

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Request portfolio data for a custom portfolio with portfolio identifier `U335877-1 Client`. Request data using all fields `f`. Filter the portfolio data by specifying the date of November 3, 2014, using the override value `REFERENCE_DATE` equal to `20141103`.

```
p = 'U335877-1 Client';
f = {'PORTFOLIO_MEMBERS','PORTFOLIO_MPOSITION',...
     'PORTFOLIO_MWEIGHT','PORTFOLIO_DATA'};
o = {'REFERENCE_DATE'};
ov = {'20141103'};

[d,plist] = portfolio(c,p,f,o,ov)
```

```
d =

    PORTFOLIO_MPOSITION: {{0x1 cell}}
      PORTFOLIO_MWEIGHT: {{0x1 cell}}
         PORTFOLIO_DATA: {{0x1 cell}}
      PORTFOLIO_MEMBERS: {{0x1 cell}}
```

```
plist =

    'U335877-1 Client'
```

`d` is a structure that contains portfolio data. Each structure field corresponds to data for each portfolio field.

`plist` is a cell array that contains the portfolio identifier.

Close the connection.

```
close(c)
```

Input Arguments

c — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

p — Portfolio

character vector | string scalar

Portfolio, specified as a character vector or string scalar. Specify the portfolio by the ID that you can find in the upper-right corner of the portfolio display page. Append the text `' Client'` (without quotes) to the ID. For example, if the ID is `U335877-1`, then specify `'U335877-1 Client'`.

Access the portfolio display page by using the **PRTU<GO>** option from the Bloomberg terminal. For details, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `'U335877-1 Client'`

Data Types: `char` | `cell` | `string`

f — Portfolio fields

`'PORTFOLIO_DATA'` | `'PORTFOLIO_MEMBERS'` | `'PORTFOLIO_MPOSITION'` | `'PORTFOLIO_MWEIGHT'`

Portfolio fields, specified as one of the preceding values for one field. To specify multiple fields, use a cell array of these values.

Bloomberg Field Name	Bloomberg Field Description
`'PORTFOLIO_DATA'`	Returns a list of the identifiers, positions, market values, cost, cost date, and cost foreign exchange rate of each security in a custom portfolio.
`'PORTFOLIO_MEMBERS'`	Returns a list of identifiers for the members of a custom portfolio.
`'PORTFOLIO_MPOSITION'`	Returns a list of identifiers and the position for each security in a custom portfolio.
`'PORTFOLIO_MWEIGHT'`	Returns a list of identifiers and the percentage weight for each security in a custom portfolio.

Data Types: `char` | `cell`

o — Bloomberg override field
character vector | string scalar | cell array of character vectors | string array

Bloomberg override field, specified as a character vector, string scalar, cell array of character vectors, or string array. The Bloomberg value `'REFERENCE_DATE'` denotes returning Bloomberg data for a specific date.

Data Types: `char` | `cell` | `string`

ov — Bloomberg override field value
`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field value, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field value. A cell array of character vectors or string array denotes multiple Bloomberg override field values. Use this field value to filter the Bloomberg data result set.

Example: `'20100101'`

Data Types: `char` | `cell` | `string`

Output Arguments

d — Portfolio data
structure (default) | table

Portfolio data, returned as a structure or table. The data type of the portfolio data depends on the DataReturnFormat property of the connection object.

`plist` — Portfolio list
cell array of character vectors

Portfolio list, returned as a cell array of character vectors for the corresponding portfolio identifiers in `p`. The contents of `plist` are identical in value and order to `p`.

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface” on page 5-28

realtime

Real-time data for Bloomberg B-PIPE connection V3

Syntax

```
d = realtime(c,s,f)
[~,t] = realtime(c,s,f,eventhandler)
```

Description

`d = realtime(c,s,f)` returns the data for the `bloombergBPIPE` object `c` with the Bloomberg B-PIPE C++ interface, security list `s`, and requested fields `f`. `realtime` accesses the Bloomberg Market Data service.

`[~,t] = realtime(c,s,f,eventhandler)` returns an empty output and the timer `t` associated with the real-time event handler for the subscription list. Given connection `c`, the `realtime` function subscribes to a security or securities `s` and requests fields `f`, to update in real time while running an event handler `eventhandler`.

Examples

Retrieve Data for One Security

Retrieve a snapshot of data for one security only.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the last trade and volume of the IBM security.

```
d = realtime(c,'IBM US Equity',{'Last_Trade','Volume'})
```

```
d =
```

```
    LAST_TRADE: '181.76'
        VOLUME: '7277793'
```

Close the Bloomberg connection.

```
close(c)
```

Retrieve Data for One Security Using Event Handler

You can create your own event handler function to process Bloomberg data. For this example, use the event handler `disp` that displays Bloomberg stock tick data at the command line.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the last price and volume for the IBM security using the event handler `disp`.

```
[~,t] = realtime(c,'IBM US Equity',{'LAST_PRICE','VOLUME'}, ...
    'disp')

t =


   Timer Object: timer-4

   Timer Settings
      ExecutionMode: fixedRate
             Period: 0.05
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: 1x5 cell array
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''

   Columns 1 through 4

    {'SecurityID'   }    {'LAST_PRICE'}    {'SecurityID'   }    {'VOLUME'}
```

```
    {'IBM US Equity'}    {'118.490000'}    {'IBM US Equity'}    {'744066'}
...
```

`realtime` returns the MATLAB timer object with its properties. Then, `realtime` returns the stock tick data for the IBM security with the last price and volume.

Stop the display of real-time data.

```
stop(t)
c.Session.stopSubscriptions
```

Close the Bloomberg connection.

```
close(c)
```

Retrieve Data for Multiple Securities Using Event Handler

You can create your own event handler function to process Bloomberg data. For this example, use the event handler `disp` that displays Bloomberg stock tick data at the command line.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the last price and volume for IBM and Ford Motor Company securities.

```
[~,t] = realtime(c,{'IBM US Equity','F US Equity'}, ...
    {'LAST_PRICE','VOLUME'},'disp')
```

```
t =


   Timer Object: timer-4

   Timer Settings
      ExecutionMode: fixedRate
              Period: 0.05
           BusyMode: drop
            Running: off

   Callbacks
```

```
           TimerFcn: 1x5 cell array
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''

Columns 1 through 6

    {'SecurityID' }    {'LAST_PRICE'}    {'SecurityID' }    {'VOLUME'  }    {'SecurityID'   }
    {'F US Equity'}    {'8.960000'  }    {'F US Equity'}    {'13423731'}    {'IBM US Equity'}

  Columns 7 through 8

    {'SecurityID'   }    {'VOLUME'}
    {'IBM US Equity'}    {'744066'}
...
```

`realtime` returns the MATLAB timer object with its properties. Then, `realtime` returns the stock tick data for the IBM and Ford Motor Company securities with the last price and volume.

Stop the display of real-time data.

```
stop(t)
c.Session.stopSubscriptions
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

f — Bloomberg data fields

character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

`eventhandler` — Event handler
character vector | string scalar

Event handler, specified as a character vector or string scalar that denotes the name of an event handler function that you define. You can define an event handler function to process any type of real-time Bloomberg events. The specified event handler function runs every time the timer fires.

Data Types: `char` | `string`

Output Arguments

`d` — Bloomberg data
structure (default) | table | timetable

Bloomberg data, returned as a structure, table, or timetable. The data type of the Bloomberg data depends on the DataReturnFormat and DatetimeType properties of the connection object. For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`t` — MATLAB timer
object

MATLAB timer, returned as a MATLAB object. For details about this object, see `timer`.

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `getdata` | `history` | `timeseries`

Topics
"Retrieve Bloomberg Real-Time Data Using Bloomberg B-PIPE C++ Interface" on page 5-37
"Writing and Running Custom Event Handler Functions" on page 1-26

tahistory

Historical technical analysis for Bloomberg B-PIPE connection V3

Syntax

```
d = tahistory(c)
d = tahistory(c,s,startdate,enddate,study,period,Name,Value)
```

Description

`d = tahistory(c)` returns the Bloomberg B-PIPE V3 session technical analysis data study and element definitions.

`d = tahistory(c,s,startdate,enddate,study,period,Name,Value)` returns the Bloomberg V3 session technical analysis data study and element definitions with additional options specified by one or more name-value pair arguments.

Examples

Request Bloomberg Directional Movement Indicator (DMI) Study for Security

Return all available Bloomberg studies and use the DMI study to run a technical analysis for a security.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

List the available Bloomberg studies.

```
d = tahistory(c)
```

```
d =

          dmiStudyAttributes: [1x1 struct]
```

```
       smavgStudyAttributes: [1x1 struct]
        bollStudyAttributes: [1x1 struct]
         maoStudyAttributes: [1x1 struct]
          fgStudyAttributes: [1x1 struct]
         rsiStudyAttributes: [1x1 struct]
        macdStudyAttributes: [1x1 struct]
         tasStudyAttributes: [1x1 struct]
       emavgStudyAttributes: [1x1 struct]
     maxminStudyAttributes: [1x1 struct]
       ptpsStudyAttributes: [1x1 struct]
       cmciStudyAttributes: [1x1 struct]
       wlprStudyAttributes: [1x1 struct]
      wmavgStudyAttributes: [1x1 struct]
    trenderStudyAttributes: [1x1 struct]
        gocStudyAttributes: [1x1 struct]
       kltnStudyAttributes: [1x1 struct]
   momentumStudyAttributes: [1x1 struct]
        rocStudyAttributes: [1x1 struct]
        maeStudyAttributes: [1x1 struct]
      hurstStudyAttributes: [1x1 struct]
       chkoStudyAttributes: [1x1 struct]
         teStudyAttributes: [1x1 struct]
      vmavgStudyAttributes: [1x1 struct]
      tmavgStudyAttributes: [1x1 struct]
        atrStudyAttributes: [1x1 struct]
        rexStudyAttributes: [1x1 struct]
        adoStudyAttributes: [1x1 struct]
         alStudyAttributes: [1x1 struct]
        etdStudyAttributes: [1x1 struct]
        vatStudyAttributes: [1x1 struct]
       tvatStudyAttributes: [1x1 struct]
         pdStudyAttributes: [1x1 struct]
         rvStudyAttributes: [1x1 struct]
     ipmavgStudyAttributes: [1x1 struct]
      pivotStudyAttributes: [1x1 struct]
         orStudyAttributes: [1x1 struct]
        pcrStudyAttributes: [1x1 struct]
         bsStudyAttributes: [1x1 struct]
```

`d` contains structures pertaining to each available Bloomberg study.

Display the name-value pairs for the DMI study.

```
d.dmiStudyAttributes
```

```
ans =

              period: [1x104 char]
     priceSourceHigh: [1x123 char]
      priceSourceLow: [1x121 char]
    priceSourceClose: [1x125 char]
```

Obtain more information about the `period` property.

```
d.dmiStudyAttributes.period
```

```
ans =

DEFINITION period {
```

```
    Min Value = 1

    Max Value = 1

    TYPE Int64

} // End Definition: period
```

Run the DMI study for the IBM security for the last month with `period` equal to `14`, the high price, the low price, and the closing price.

```
d = tahistory(c,'IBM US Equity',floor(now)-30,floor(now),'dmi',...
              'all_calendar_days','period',14,...
              'priceSourceHigh','PX_HIGH',...
              'priceSourceLow','PX_LOW','priceSourceClose','PX_LAST')

d =

         date: [31x1 double]
     DMI_PLUS: [31x1 double]
    DMI_MINUS: [31x1 double]
          ADX: [31x1 double]
         ADXR: [31x1 double]
```

`d` contains a `studyDataTable` with one `studyDataRow` for each interval returned.

Display the first five dates in the returned data.

```
d.date(1:5,1)

ans =

     735507.00
     735508.00
     735509.00
     735510.00
     735511.00
```

Display the first five prices in the plus DI line.

```
d.DMI_PLUS(1:5,1)

ans =

         18.92
         17.84
         16.83
         15.86
         15.63
```

Display the first five prices in the minus DI line.

```
d.DMI_MINUS(1:5,1)

ans =

         30.88
         29.12
```

```
         28.16
         30.67
         29.24
```

Display the first five values of the Average Directional Index.

```
d.ADX(1:5,1)
```

```
ans =

         22.15
         22.28
         22.49
         23.15
         23.67
```

Display the first five values of the Average Directional Movement Index Rating.

```
d.ADXR(1:5,1)
```

```
ans =

         25.20
         25.06
         25.05
         25.60
         26.30
```

Close the Bloomberg connection.

```
close(c)
```

Request DMI Study for Security with Pricing Source

Run a technical analysis to return the DMI study for a security with a pricing source.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Run the DMI study for the Microsoft security with pricing source `ETPX` for the last month with `period` equal to `14`, the high price, the low price, and the closing price.

```
d = tahistory(c,'MSFT@ETPX US Equity',floor(now)-30,floor(now),...
              'dmi','all_calendar_days','period',14,...
              'priceSourceHigh','PX_HIGH','priceSourceLow','PX_LOW',...
              'priceSourceClose','PX_LAST')
```

```
d =

         date: [31x1 double]
     DMI_PLUS: [31x1 double]
    DMI_MINUS: [31x1 double]
          ADX: [31x1 double]
         ADXR: [31x1 double]
```

`d` contains a `studyDataTable` with one `studyDataRow` for each interval returned.

Display the first five dates in the returned data.

```
d.date(1:5,1)
```

```
ans =

     735507.00
     735508.00
     735509.00
     735510.00
     735511.00
```

Display the first five prices in the plus DI line.

```
d.DMI_PLUS(1:5,1)
```

```
ans =

         28.37
         30.63
         32.72
         30.65
         29.37
```

Display the first five prices in the minus DI line.

```
d.DMI_MINUS(1:5,1)
```

```
ans =

         21.97
         21.17
         19.47
         18.24
         17.48
```

Display the first values of the Average Directional Index.

```
d.ADX(1:5,1)
```

```
ans =

         13.53
         13.86
```

```
         14.69
         15.45
         16.16
```

Display the first five values of the Average Directional Movement Index Rating.

```
d.ADXR(1:5,1)
```

```
ans =

         15.45
         15.36
         15.53
         15.85
         16.37
```

Close the Bloomberg connection.

```
close(c)
```

Return DMI Study Data as Table with Dates

Create a Bloomberg connection, and then return data for a DMI study. The `tahistory` function returns data for dates as a `datetime` array.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `tahistory` function returns data as a structure.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Run the DMI study for the IBM security from June 12, 2017, through June 16, 2017, with `period` equal to 14, the high price, the low price, and the closing price.

```
d = tahistory(c,'IBM US Equity','6/12/2017','6/16/2017','dmi', ...
    'all_calendar_days','period',14,'priceSourceHigh','PX_HIGH', ...
    'priceSourceLow','PX_LOW','priceSourceClose','PX_LAST');
```

Access the DMI study data for the first three dates.

```
d(1:3,:)
```

```
ans =

  3×5 table

       date        DMI_PLUS    DMI_MINUS     ADX     ADXR
    ___________    ________    _________    _____    _____

    12-Jun-2017     30.48        16.31      33.93    45.26
    13-Jun-2017     28.88        15.45      33.67    44.10
    14-Jun-2017     26.62        18.98      32.46    42.67
```

`d` is a `table` that contains these columns:

- `date` -- Date
- `DMI_PLUS` -- Prices in plus DI line
- `DMI_MINUS` -- Prices in minus DI line
- `ADX` -- Average Directional Index values
- `ADXR` -- Average Directional Movement Index Rating values

Access the first three dates in the returned data.

```
d.date(1:3)
```

```
ans =

  3×1 datetime array

   12-Jun-2017
   13-Jun-2017
   14-Jun-2017
```

Close the Bloomberg connection.

```
close(c)
```

Return DMI Study Data as Timetable

Create a Bloomberg connection, and then return data for a DMI study. The `tahistory` function returns data as a `timetable`.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `tahistory` function returns data as a structure.

```
c.DataReturnFormat = 'timetable';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Run the DMI study for the IBM security from June 12, 2017 through June 16, 2017 with `period` equal to 14, the high price, the low price, and the closing price.

```
d = tahistory(c,'IBM US Equity','6/12/2017','6/16/2017','dmi', ...
    'all_calendar_days','period',14,'priceSourceHigh','PX_HIGH', ...
    'priceSourceLow','PX_LOW','priceSourceClose','PX_LAST');
```

Access the DMI study data for the first three dates.

```
d(1:3,:)
```

```
ans =

  3×4 timetable

       date        DMI_PLUS    DMI_MINUS     ADX     ADXR
    ___________    ________    _________    _____    _____

    12-Jun-2017     30.48        16.31      33.93    45.26
    13-Jun-2017     28.88        15.45      33.67    44.10
    14-Jun-2017     26.62        18.98      32.46    42.67
```

`d` is a `timetable` that contains these columns:

- `date` -- Date
- `DMI_PLUS` -- Prices in plus DI line
- `DMI_MINUS` -- Prices in minus DI line

- `ADX` -- Average Directional Index values
- `ADXR` -- Average Directional Movement Index Rating values

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

`s` — Security

character vector | string scalar

Security, specified as a character vector or string scalar for a single Bloomberg security.

Data Types: `char` | `string`

`startdate` — Start date

numeric scalar | character vector | string scalar

Start date, specified as a numeric scalar, character vector, or string scalar to denote the start date of the date range for the returned tick data.

Example: `floor(now-1)`

Data Types: `double` | `char` | `string`

`enddate` — End date

numeric scalar | character vector | string scalar

End date, specified as a numeric scalar, character vector, or string scalar to denote the end date of the date range for the returned tick data.

Example: `floor(now)`

Data Types: `double` | `char` | `string`

`study` — Study type

character vector | string scalar

Study type, specified as a character vector or string scalar to denote the study to use for historical analysis.

Data Types: `char` | `string`

`period` — Periodicity

`'daily'` | `'weekly'` | `'monthly'` | `'quarterly'` | ...

Periodicity, specified as one of these values to denote the data to return. For specifying multiple values, use a cell array. For example, when `period` is set to `{'daily','all_calendar_days'}`, `tahistory` returns daily data for all calendar days, and reports missing data as `NaN`s. When `period` is set to `'active_days_only'`, `tahistory` returns data using the default periodicity for active

trading days only. The default periodicity depends on the security. If a security is reported on a monthly basis, the default periodicity is monthly. These tables show the values for `period`.

To specify the periodicity of the return data, see this table.

Value	Description
`'daily'`	Return data for each day.
`'weekly'`	Return data for each week.
`'monthly'`	Return data for each month.
`'quarterly'`	Return data for each quarter.
`'semi_annually'`	Return data semiannually.
`'yearly'`	Return data for each year.

The anchor date is the date to which all other reported dates are related. To specify the anchor date, see this table.

Value	Description
`'actual'`	Anchor date specification for an actual date. For this function, for periodicities other than daily, `enddate` is the anchor date. If the period is weekly and the `enddate` is a Thursday, every data point is a Thursday, or the nearest prior business day to Thursday. If the period is monthly and the `enddate` is the 20th of a month, every data point is the 20th of each month in the date range.
`'calendar'`	Anchor date specification for a calendar year.
`'fiscal'`	Anchor date specification for a fiscal year.

To specify returning data for particular days, see this table.

Value	Description
`'non_trading_weekdays'`	Return data for all weekdays.
`'all_calendar_days'`	Return data for all calendar days.
`'active_days_only'`	Return data for only active trading days.

To specify how to fill missing values, see this table.

Value	Description
`'previous_value'`	Fill missing values with previous values for dates without trading activity for the security.
`'nil_value'`	Fill missing values with a `NaN` for dates without trading activity for the security.

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'period',14`, `'priceSourceHigh','PX_HIGH'`, `'priceSourceLow','PX_LOW'`, `'priceSourceClose','PX_LAST'`

Note For details about the full list of name-value pair arguments, see the Bloomberg tool located at `C:\blp\API\APIv3\bin\BBAPIDemo.exe`.

`period` — Period
numeric scalar

Period, specified as the comma-separated pair consisting of `'period'` and a numeric scalar. For details about the period, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `double`

`priceSourceHigh` — High price
character vector | string scalar

High price, specified as the comma-separated pair consisting of `'priceSourceHigh'` and a character vector or string scalar. For details about the high price, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

`priceSourceLow` — Low price
character vector | string scalar

Low price, specified as the comma-separated pair consisting of `'priceSourceLow'` and a character vector or string scalar. For details about the low price, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

`priceSourceClose` — Closing price
character vector | string scalar

Closing price, specified as the comma-separated pair consisting of `'priceSourceClose'` and a character vector or string scalar. For details about the closing price, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

Output Arguments

`d` — Technical analysis data
structure (default) | table | timetable

Technical analysis data, returned as a structure, table, or timetable. The data type of the returned data depends on the DataReturnFormat and DatetimeType properties of the connection object.

For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

"Retrieve Bloomberg Current Data Using Bloomberg B-PIPE C++ Interface" on page 5-28

timeseries

Intraday tick data for Bloomberg B-PIPE connection V3

Syntax

```
d = timeseries(c,s,date)
d = timeseries(c,s,date,interval,field)
d = timeseries(c,s,date,[],field,options,values)

d = timeseries(c,s,{startdate,enddate})
d = timeseries(c,s,{startdate,enddate},interval,field)
d = timeseries(c,s,{startdate,enddate},[],field)
d = timeseries(c,s,{startdate,enddate},[],field,options,values)

d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval)
d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval,field)

d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime},
interval)
d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime},
interval,field)
```

Description

`d = timeseries(c,s,date)` retrieves raw tick data using the `bloombergBPIPE` object with the Bloomberg B-PIPE C++ interface and a security for a specific date.

`d = timeseries(c,s,date,interval,field)` retrieves raw tick data that is aggregated into intervals for a specific field.

`d = timeseries(c,s,date,[],field,options,values)` retrieves raw tick data without an aggregation interval for a specific field with the specified options and corresponding values.

`d = timeseries(c,s,{startdate,enddate})` retrieves raw tick data for a date range using a start date and an end date.

`d = timeseries(c,s,{startdate,enddate},interval,field)` retrieves raw tick data for a specific date range aggregated into intervals for a specific field.

`d = timeseries(c,s,{startdate,enddate},[],field)` retrieves raw tick data for a specific date range without an aggregation interval for a specific field.

`d = timeseries(c,s,{startdate,enddate},[],field,options,values)` retrieves raw tick data for a specific date range without an aggregation interval for a specific field with specified options and corresponding values.

`d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval)` retrieves raw trade tick data for a specific time range for each day within a specific date range, aggregated into intervals.

`d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval,field)` uses a specific field for tick data to return.

`d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime}, interval)` retrieves raw trade tick data for a whole day increment within a specific date and time range, aggregated into intervals.

`d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime}, interval,field)` uses a specific field for tick data to return.

Examples

Retrieve Tick Data for Specific Date and Pricing Source

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date. Use a security with and without a pricing source to retrieve tick data.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the trade tick series using the IBM security for today.

```
d = timeseries(c,'IBM US Equity',floor(now))
```

```
d =

    'TRADE'    [735537.40]    [181.69]    [100.00]
    'TRADE'    [735537.40]    [181.69]    [100.00]
    'TRADE'    [735537.40]    [181.68]    [100.00]
    ...
```

The columns in `d` are:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size

Here, the first row shows that 100 IBM shares sold for $181.69 today.

Retrieve the trade tick series using the Microsoft security with pricing source ETPX for today.

```
d = timeseries(c,'MSFT@ETPX US Equity',floor(now))
```

```
d =

    'TRADE'    [735537.40]    [35.53]    [100.00]
    'TRADE'    [735537.40]    [35.55]    [200.00]
    'TRADE'    [735537.40]    [35.55]    [100.00]
    ...
```

Here, the first row shows that 100 Microsoft shares are sold for $35.53 today.

Close the Bloomberg connection.

```
close(c)
```

Time Interval with Specific Field

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date. Specify the tick data to return using a time interval and field.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the trade tick series using the IBM security aggregated into 5-minute intervals for today.

```
d = timeseries(c,'IBM US Equity',floor(now),5,'Trade')
```

```
d =

  Columns 1 through 7

   735537.40        181.69        181.99        180.10        181.84     252322.00        861.00
   735537.40        181.90        181.97        181.57        181.65      78570.00        535.00
   735537.40        181.73        182.18        181.58        182.07     124898.00        817.00
     ...

  Column 8

   45815588.00
   14282076.00
```

```
   22710954.00
   ...
```

The columns in `d` contain the following:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

Here, the first row of data shows prices and tick data for the current date. The next row shows tick data for 5 minutes later.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Tick Data Using Option and Value

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date and field. Use option and value to return additional data.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the trade tick series using the `'F US Equity'` security without specifying the aggregation parameter for today. Also, return the condition codes.

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               'includeConditionCodes','true')

d =
```

```
    'TRADE'    [735556.57]    [17.12]    [   100.00]    'R6,IS'
    'TRADE'    [735556.57]    [17.12]    [   100.00]    ''
    'TRADE'    [735556.57]    [17.12]    [   500.00]    ''
    ...
```

The columns in `d` contain the following:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size
- Condition codes

Here, the first row shows that 100 `'F US Equity'` security shares sold for $17.12 today.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Tick Data Using Date Range

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the tick series for the `'F US Equity'` security for the last business day from the beginning of the day to noon.

```
d = timeseries(c,'F US Equity',{floor(now-4),floor(now-3.5)})
```

```
d =

    'TRADE'    [735552.67]    [17.09]    [   200.00]
    'TRADE'    [735552.67]    [17.09]    [   100.00]
    'TRADE'    [735552.67]    [17.09]    [   100.00]
   ...
```

The columns in `d` are:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size

Here, the first row shows that 200 `'F US Equity'` security shares were sold for $17.09 on the last business day.

Close the Bloomberg connection.

```
close(c)
```

Date Range with Interval and Specific Field

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range. Specify the interval and field.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the trade tick series for the past 50 days for the IBM security aggregated into 5-minute intervals.

```
d = timeseries(c,'IBM US Equity',{floor(now)-50,floor(now)},5,'Trade')
```

```
ans =

  Columns 1 through 7

   735487.40        187.20        187.60        187.02        187.08     207683.00        560.00
   735487.40        187.03        187.13        186.65        186.78      46990.00        349.00
   735487.40        186.78        186.78        186.40        186.47      51589.00        399.00
     ...

Column 8

   38902968.00
    8779374.00
    9626896.00
   ...
```

The columns in `d` contain the following:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row of data shows prices and tick data for the current date. The next row shows tick data for 5 minutes later.

Close the Bloomberg connection.

```
close(c)
```

Date Range with Numerous Fields

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range and numerous fields.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return the Bid, Ask, and trade tick series for the security `'F US Equity'` for yesterday with a time interval at noon, without specifying the aggregation parameter.

```
d = timeseries(c,'F US Equity',{floor(now-1)+.5,floor(now-1)+.51},...
               [],{'Bid','Ask','Trade'})
```

```
d =

    'TRADE'    [735550.50]    [16.71]    [100.00]
    'ASK'      [735550.50]    [16.71]    [312.00]
```

```
    'BID'      [735550.50]    [16.70]    [177.00]
    ...
```

The columns in `d` are:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size

Here, the first row shows that 100 `'F US Equity'` security shares sold for $16.71 yesterday.

Close the Bloomberg connection.

```
close(c)
```

Date Range with Options and Values

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range. Specify options and values to return additional data.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return the trade tick series for the security `'F US Equity'` for yesterday with a time interval at noon, without specifying the aggregation parameter. Also, return the condition codes, exchange codes, and broker codes.

```
d = timeseries(c,'F US Equity',{floor(now-1)+.5,floor(now-1)+.51},...
               [],'Trade',{'includeConditionCodes',...
               'includeExchangeCodes','includeBrokerCodes'},...
               {'true','true','true'})

d =

    'TRADE'    [735550.50]    [16.71]    [100.00]    'T'     'D'
    'TRADE'    [735550.50]    [16.70]    [400.00]    'IS'    'B'
    'TRADE'    [735550.50]    [16.70]    [100.00]    'IS'    'B'
    ...
```

The columns in `d` contain the following:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size
- Exchange condition codes
- Exchange codes

Broker codes are available for Canadian, Finnish, Mexican, Philippine, and Swedish equities only. In this case, the broker buy code appears in the seventh column and the broker sell code appears in the eighth column.

Here, the first row shows that 100 `'F US Equity'` security shares sold for $16.71 yesterday.

Close the Bloomberg connection.

```
close(c)
```

Date and Time Range with Interval

Use Bloomberg to retrieve raw trade tick data by specifying a time range for each day in a specific date range. Specify the time interval for the tick data.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the trade tick series for the `'F US Equity'` security for the last two days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. `d` is a numeric matrix.

```
s = 'F US Equity';
startdate = datetime('today')-1;
enddate = datetime('today');
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;
```

```
d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

     736959.40         11.71         11.81         11.71         11.79
     736959.40         11.79         11.81         11.75         11.79
     736959.40         11.80         11.82         11.78         11.80

  Columns 6 through 8

    1375547.00       1190.00   16196757.00
     598924.00        898.00    7058724.00
     488655.00        641.00    5768371.50
```

The columns in d are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

Determine the maximum high price for the last two days.

```
highprices = d(:,3);
m = max(highprices)
```

```
m =

         11.82
```

Close the Bloomberg connection.

```
close(c)
```

Date and Time Range with Interval and Specific Field

Use Bloomberg to retrieve raw tick data by specifying a time range for each day in a specific date range. Specify the time interval and the field for the type of tick data to return. Here, specify the bid tick data.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the tick series for the `'F US Equity'` security for the last two days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. Specify retrieving the bid tick series. `d` is a numeric matrix.

```
s = 'F US Equity';
startdate = datetime('today')-1;
enddate = datetime('today');
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;
field = 'BID';

d = timeseries(c,s, ...
    {startdate:enddate,starttime,endtime}, ...
    interval,field);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5
```

```
   736959.40         11.70         11.80         11.70         11.79
   736959.40         11.79         11.80         11.75         11.79
   736959.40         11.79         11.81         11.78         11.80

Columns 6 through 8

   397711.00       1442.00    4681704.50
   450997.00       1698.00    5311330.50
   464761.00       1391.00    5481707.50
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

Determine the maximum high price for the last two days.

```
highprices = d(:,3);
m = max(highprices)
```

```
m =

         11.81
```

Close the Bloomberg connection.

```
close(c)
```

Date and Time Range with Day Increment and Interval

Use Bloomberg to retrieve raw trade tick data by specifying a time range for each day in a specific date range. Specify a day increment for the date range and the time interval for the tick data.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.

- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the trade tick series for the `'IBM US Equity'` security for the last two months. Set the day increment to 5 days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. `d` is a numeric matrix.

```
s = 'IBM US Equity';
startdate = datetime('today')-60;
enddate = datetime('today');
dayincrement = 5;
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;

d = timeseries(c,s, ...
    {startdate:dayincrement:enddate,starttime,endtime}, ...
    interval);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    736900.40        147.00        147.04        146.55        146.62
    736900.40        146.62        146.87        146.62        146.71
    736900.40        146.72        146.79        146.52        146.54

  Columns 6 through 8

    125558.00        393.00   18440146.00
     39535.00        258.00    5800969.00
     49659.00        314.00    7282961.00
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price

- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

After the tick data for the first day in the date range, `d` contains tick data for a trading day that is 5 days later.

Close the Bloomberg connection.

```
close(c)
```

Date and Time Range with Day Increment, Interval, and Specific Field

Use Bloomberg to retrieve raw tick data by specifying a time range for each day in a specific date range. Specify a day increment for the date range, the time interval for the tick data, and the field for the type of tick data to return. Here, specify the bid tick data.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Retrieve the trade tick series for the `'F US Equity'` security for the last two months. Set the day increment to 5 days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. Specify the bid tick series. `d` is a numeric matrix.

```
s = 'F US Equity';
startdate = datetime('today')-60;
enddate = datetime('today');
dayincrement = 5;
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;
```

```
field = 'BID';

d = timeseries(c,s, ...
    {startdate:dayincrement:enddate,starttime,endtime}, ...
    interval,field);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

     736900.40         11.50         11.54         11.49         11.50
     736900.40         11.50         11.50         11.48         11.48
     736900.40         11.48         11.49         11.44         11.44

  Columns 6 through 8

     422305.00       1158.00    4863894.00
     575966.00       1180.00    6617854.00
     288147.00       1489.00    3305491.75
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

After the tick data for the first day in the date range, `d` contains tick data for a trading day that is 5 days later.

Close the Bloomberg connection.

```
close(c)
```

Return Tick Data as Table with Dates

Create a Bloomberg connection, and then return intraday tick data. The `timeseries` function returns data for dates as a `datetime` array.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `timeseries` function returns data as a numeric array.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve the trade tick series for the IBM® security aggregated into 5-minute intervals for today. `d` is a table that contains the tick series data.

```
s = 'IBM US Equity';
date = floor(now);
interval = 5;
field = 'Trade';

d = timeseries(c,s,date,interval,field);
```

Access the first three ticks of data.

```
d(1:3,:)
```

```
ans =

  3×8 table

        DATE          OPEN       HIGH       LOW       CLOSE      VOLUME      NUMBER_OF_TICKS     TOTAL_
    ____________    _______    _______    _______    _______    _________    _______________    ______
```

```
    21-Dec-2017    153.17    153.31    153.08    153.31    152524.00        442.00        23367
    21-Dec-2017    153.35    153.35    152.82    152.84     46051.00        291.00         7048
    21-Dec-2017    152.84    153.21    152.82    153.16     30966.00        225.00         4737
```

`d` contains columns with the following data:

- Date
- Open price
- High price
- Low price
- Closing price
- Volume
- Number of ticks
- Total tick value in the bar

Access the first three dates in the `DATE` column.

```
d.DATE(1:3)
```

```
ans =

  3×1 datetime array

   21-Dec-2017
   21-Dec-2017
   21-Dec-2017
```

Close the Bloomberg connection.

```
close(c)
```

Return Tick Data as Timetable

Create a Bloomberg connection, and then return intraday tick data. The `timeseries` function returns data for dates as a `timetable`.

Create a Bloomberg B-PIPE connection using the IP address of the machine running the Bloomberg B-PIPE process. This example uses the Bloomberg B-PIPE C++ interface and assumes the following:

- The authentication is Windows authentication when you set `authtype` to `'OS_LOGON'`.
- The application name is blank because you are not connecting to Bloomberg B-PIPE using an application.
- The IP address for the machine running the Bloomberg B-PIPE process is `'111.11.11.112'`.
- The port number of the machine running the Bloomberg B-PIPE process is `8194`.

`c` is a `bloombergBPIPE` object.

```
authtype = 'OS_LOGON';
appname = '';
ipaddress = {'111.11.11.112'};
port = 8194;

c = bloombergBPIPE(authtype,appname,ipaddress,port);
```

Return data as a `timetable` by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `timeseries` function returns data as a numeric array.

```
c.DataReturnFormat = 'timetable';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve the trade tick series for the IBM® security aggregated into 5-minute intervals for today. `d` is a `timetable` that contains the tick series data.

```
s = 'IBM US Equity';
date = floor(now);
interval = 5;
field = 'Trade';

d = timeseries(c,s,date,interval,field);
```

Access the first three ticks of data.

```
d(1:3,:)
```

```
ans =

  3×7 timetable

      DATE         OPEN      HIGH      LOW      CLOSE      VOLUME      NUMBER_OF_TICKS    TOTAL_
    ___________    ______    ______    ______    ______    _________    _______________    ______

    21-Dec-2017    153.17    153.31    153.08    153.31    152524.00        442.00         23367
    21-Dec-2017    153.35    153.35    152.82    152.84     46051.00        291.00          7048
    21-Dec-2017    152.84    153.21    152.82    153.16     30966.00        225.00          4737
```

`d` is a `timetable` that contains the following data:

- Date
- Open price
- High price
- Low price
- Closing price
- Volume
- Number of ticks
- Total tick value in the bar

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg B-PIPE connection

`bloombergBPIPE` object

Bloomberg B-PIPE connection, specified as a `bloombergBPIPE` object.

`s` — Security

character vector | string scalar

Security, specified as a character vector or string scalar for a single Bloomberg security.

Data Types: `char` | `string`

`date` — Date

numeric scalar | character vector | string scalar | `datetime` array

Date, specified as a numeric scalar, character vector, string scalar, or `datetime` array. `date` specifies the date for the returned tick data based on the entire day from midnight until 11:59:59 p.m.

Example: `floor(now)`

Data Types: `double` | `char` | `string` | `datetime`

`interval` — Time interval

numeric scalar

Time interval, specified as a numeric scalar to denote the number of minutes between ticks for the returned tick data.

Data Types: `double`

`field` — Bloomberg field

`'TRADE'` (default) | `'BID'` | `'ASK'` | ...

Bloomberg field, specified as one of these values that define the tick data to return.

Request Type	Valid Bloomberg Field Values
IntradayBarRequest with time interval specified	`'TRADE'`
	`'BID'`
	`'ASK'`
	`'BID_BEST'`
	`'ASK_BEST'`
IntradayTickRequest with no time interval specified	`'TRADE'`
	`'BID'`
	`'ASK'`
	`'BID_BEST'`
	`'ASK_BEST'`
	`'SETTLE'`

options — Bloomberg API options
'includeConditionCodes' | 'includeExchangeCodes' | 'includeBrokerCodes' | ...

Bloomberg API options, specified as one of the values in this table.

Value	Description
'includeConditionCodes'	Exchange condition codes associated with the event
'includeExchangeCodes'	Exchange code where tick originated
'includeBrokerCodes'	Broker code
'includeRpsCodes'	Reporting party side
'includeNonPlottableEvents'	After-hours data

Note The value 'includeNonPlottableEvents' applies to raw intraday requests only.

To specify more than one Bloomberg API option, use a cell array of these values.

Specify the corresponding Bloomberg API value for each API option. The number of options must match the number of values.

For example, to specify one Bloomberg API option, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               'includeConditionCodes','true');
```

To specify two Bloomberg API options, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               {'includeConditionCodes','includeExchangeCodes'},...
               {'true','true'});
```

For details about the options, see the *Bloomberg API Developer's Guide*.

Data Types: char | cell

values — Bloomberg API values
'true' | 'false'

Bloomberg API values, specified as 'true' or 'false'. Each value corresponds to the specified Bloomberg API option. To specify more than one Bloomberg API value, use a cell array. The number of values must match the number of options.

For example, to specify one Bloomberg API option, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               'includeConditionCodes','true');
```

To specify two Bloomberg API options, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               {'includeConditionCodes','includeExchangeCodes'},...
               {'true','true'});
```

Data Types: `char` | `cell`

`startdate` — Start date
numeric scalar | character vector | string scalar | `datetime` array

Start date, specified as a numeric scalar, character vector, string scalar, or `datetime` array. This date specifies the beginning of the date range for the returned tick data. If no ticks are present in the date range, then returned tick data is empty.

Example: `floor(now-1)`

Data Types: `double` | `char` | `string` | `datetime`

`enddate` — End date
numeric scalar | character vector | string scalar | `datetime` array

End date, specified as a numeric scalar, character vector, string scalar, or `datetime` array. This date specifies the end of the date range for the returned tick data. If no ticks are present in the date range, then returned tick data is empty.

Example: `floor(now)`

Data Types: `double` | `char` | `string` | `datetime`

`starttime` — Start time
character vector | string scalar | `datetime` array

Start time, specified as a character vector, string scalar, or `datetime` array. This time specifies the start time of the time range for the returned tick data.

Example: `'09:30:00'`

Data Types: `char` | `string` | `datetime`

`endtime` — End time
character vector | string scalar | `datetime` array

End time, specified as a character vector, string scalar, or `datetime` array. This time specifies the end time of the time range for the returned tick data.

Example: `'16:30:00'`

Data Types: `char` | `string` | `datetime`

`dayincrement` — Day increment
1 (default) | numeric scalar

Day increment, specified as a numeric scalar. This number specifies the whole day increment for a specific date range. For example, if the day increment is 7, then the returned data contains ticks for every 7th day starting from the first day within the date range.

Data Types: `double`

Output Arguments

`d` — Bloomberg tick data
cell array | numeric array | table | timetable

Bloomberg tick data, returned as one of these data types:

- Cell array for requests without a specified time interval (raw tick data)
- Numeric array for requests with a specified time interval
- table
- timetable

The data type of the tick data depends on the DataReturnFormat and DatetimeType properties of the connection object.

Note The Bloomberg API returns the tick time with precision in seconds.

Limitations

When the data request is too large, `timeseries` displays this error message:

```
Timeout error:
Error using blp/timeseries>processResponseEvent (line 338) REQUEST FAILED: responseError = {

source = bbdbl7

code = -2

category = TIMEOUT

message = Timed out getting data from store [nid:327]

subcategory = INTERNAL_ERROR

}
```

To fix this error, shorten the length of the date range by modifying the input arguments `startdate` and `enddate`.

Tips

- You cannot retrieve Bloomberg intraday tick data for a date more than 140 days ago.
- The *Bloomberg API Developer's Guide* states that `'TRADE'` corresponds to LAST_PRICE for IntradayTickRequest and IntradayBarRequest.
- Bloomberg V3 intraday tick data supports additional name-value pairs. For details on these pairs, see the *Bloomberg API Developer's Guide* by typing `WAPI` and clicking the **<GO>** button on the Bloomberg terminal.
- You can check data and field availability by using the Bloomberg Excel Add-In.

Version History

Introduced in R2021a

See Also

`bloombergBPIPE` | `close` | `getdata` | `history` | `realtime`

Topics

"Retrieve Bloomberg Intraday Tick Data Using Bloomberg B-PIPE C++ Interface" on page 5-35

bloombergServer

Bloomberg Server connection V3

Description

The `bloombergServer` function creates a `bloombergServer` object. The `bloombergServer` object represents a Bloomberg Server connection using the Bloomberg V3 C++ API.

Other Datafeed Toolbox functions connect to different Bloomberg services: Bloomberg Desktop (`bloomberg`) and Bloomberg B-PIPE (`bloombergBPIPE`). For details about these services, see “Comparing Bloomberg Connections” on page 2-4.

For details about Bloomberg connection requirements, see “Data Server Connection Requirements” on page 1-3. To ensure a successful Bloomberg connection, perform the required steps before executing `bloombergServer`. For details, see “Installing Bloomberg and Configuring Connections” on page 1-5.

Creation

Syntax

```
c = bloombergServer(uuid,ipaddress)
c = bloombergServer(uuid,ipaddress,port)
c = bloombergServer(uuid,ipaddress,port,timeout)
```

Description

`c = bloombergServer(uuid,ipaddress)` creates a Bloomberg Server connection object `c` to the Bloomberg Server running on another machine, and sets the Uuid and IPAddress properties. You need a Bloomberg Server software license for the machine running the Bloomberg Server.

`c = bloombergServer(uuid,ipaddress,port)` also sets the port property.

`c = bloombergServer(uuid,ipaddress,port,timeout)` also sets the timeout property.

Caution To refer to a Bloomberg connection in other functions, use the connection object created by the `bloombergServer` function. Otherwise, using `bloombergServer` as an input argument opens multiple Bloomberg connections, causing unexpected behavior and exhausting memory resources.

Properties

`Uuid` — Bloomberg user identity UUID

numeric scalar

Bloomberg user identity UUID, specified as a numeric scalar. To find your UUID, enter `IAM` in the Bloomberg terminal and press **GO**.

Example: `12345678`

Data Types: `double`

`User` — Bloomberg user
Bloomberg user identity object

This property is read-only.

Bloomberg user, specified as a Bloomberg user identity object.

Example: `[1x1 com.bloomberglp.blpapi.impl.aT]`

`Userip` — IP address of the machine running MATLAB
character vector

This property is read-only.

IP address of the machine running MATLAB, specified as a character vector.

Example: `'111.11.11.111'`

Data Types: `char`

`Session` — Bloomberg V3 session
Bloomberg V3 API Session object

This property is read-only.

Bloomberg V3 session, specified as a Bloomberg V3 API Session object.

Example: `[1x1 BLPSession]`

`IPAddress` — Bloomberg Server IP address
character vector | string scalar

Bloomberg Server IP address, specified as a character vector or string scalar that identifies the machine running the Bloomberg Server.

Example: `'111.11.11.111'`

Data Types: `char` | `string`

`Port` — Port number
numeric scalar

Port number, specified as a numeric scalar that identifies the port number of the machine running the Bloomberg Server.

Example: `8194`

Data Types: `double`

`TimeOut` — Timeout
numeric scalar

Timeout specifying the time in milliseconds that MATLAB attempts to connect to the machine running the Bloomberg Server before timing out, specified as a numeric scalar.

Example: `10`

Data Types: `double`

`DatetimeType` — Date and time data type

`''` (default) | `'datetime'`

Date and time data type, specified as one of these values.

Value	Description
`''` (default)	Return date and time values as MATLAB date numbers.
`'datetime'`	Return date and time values as a `datetime` array.

You can specify these values using a character vector or string (for example, `"datetime"`).

When you create a `bloombergServer` object, the `bloombergServer` function leaves this property unset. To retrieve data, you must set this property value manually at the command line or in a script using dot notation, for example:

```
c.DatetimeType = 'datetime';
```

Then, you can use these supported functions:

- getbulkdata
- getdata
- history
- tahistory
- timeseries

> **Note** If the `DataReturnFormat` property value is `'table'` and the `DatetimeType` property value is `'datetime'`, then the returned data is a table that contains date and time values as a `datetime` array. If the `DataReturnFormat` property value is an empty character vector, then setting the `DatetimeType` property to `'datetime'` returns date and time values for aggregated ticks and historical requests as MATLAB date numbers.

`DataReturnFormat` — Data return format

`'cell'` | `'structure'` | `'table'` | `'timetable'`

Data return format, specified as one of these values, which determine the data type of the returned data.

Value	Data Type of Returned Data
`'cell'`	`cell array`
`'table'`	`table`

Value	Data Type of Returned Data
`'timetable'`	`timetable`
`'structure'`	`structure`

Note The default data type of the returned data depends on the executed function. To specify the default data type, set the `DataReturnFormat` property to `''`. For default data types, see the supported function list.

You can specify these values using a character vector or string (for example, `"table"`).

When you create a `bloombergServer` object, the `bloombergServer` function leaves this property unset. To retrieve data, you must set this property value manually at the command line or in a script using dot notation, for example:

```
c.DataReturnFormat = 'structure';
```

Then, you can use these supported functions.

Supported Function	Valid Data Types for Returned Data
category	• `cell array` (default) • `structure` • `table`
eqs	• `cell array` (default) • `structure` • `table`
fieldinfo	• `cell array` (default) • `structure` • `table`
fieldsearch	• `cell array` (default) • `structure` • `table`
lookup	• `structure` (default) • `table`
portfolio	• `structure` (default) • `table`

Supported Function	Valid Data Types for Returned Data
getbulkdata	• `structure` (default) • `table` • `timetable`
getdata	• `structure` (default) • `table` • `timetable`
history	• numeric array (default) • `table` • `timetable`
tahistory	• `structure` (default) • `table` • `timetable`
timeseries	• `cell array` (default for raw tick data) • numeric array (default for interval tick data) • `table` • `timetable`

Note Regardless of the `DatetimeType` property value, if the `DataReturnFormat` property value is `'timetable'`, then the `getdata` and `getbulkdata` functions return a table that contains date and time values as `datetime` arrays.

Object Functions

Bloomberg Server Connection

close Close Bloomberg Server connection V3
isconnection Determine Bloomberg Server connection V3

Bloomberg Server Data Retrieval

eqs Equity screening data for Bloomberg Server connection V3
get Properties of Bloomberg Server connection V3
getbulkdata Bulk data with header information for Bloomberg Server connection V3
getdata Current data for Bloomberg Server connection V3
history Historical data for Bloomberg Server connection V3
portfolio Current portfolio data for Bloomberg Server connection V3
realtime Real-time data for Bloomberg Server connection V3
tahistory Historical technical analysis for Bloomberg Server connection V3
timeseries Intraday tick data for Bloomberg Server connection V3

Retrieve Bloomberg Server Information

category	Field category search for Bloomberg Server connection V3
fieldinfo	Field information for Bloomberg Server connection V3
fieldsearch	Field search for Bloomberg Server connection V3
lookup	Find information about securities for Bloomberg Server connection V3

Examples

Connect to Bloomberg Server

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress)
```

```
c =

  bloombergServer with properties:

               Uuid: 12345678
               User: []
             Userip: '111.11.11.112'
            Session: [1×1 BLPSession]
          IPAddress: '111.11.11.111'
               Port: 8194
            TimeOut: 0
       DatetimeType: ''
    DataReturnFormat: ''
```

The `bloombergServer` function connects to the machine running the Bloomberg Server using the default port number `8194`. The `bloombergServer` function creates the `bloombergServer` object `c` with these properties:

- Bloomberg user identity UUID
- Bloomberg user identity object
- IP address of the machine running MATLAB
- Bloomberg V3 API Session object
- IP address of the machine running the Bloomberg Server
- Port number of the machine running the Bloomberg Server
- Number in milliseconds specifying how long MATLAB attempts to connect to the machine running the Bloomberg Server before timing out
- Date and time data type
- Data return format

Request the last and open prices for Microsoft.

```
format bank  % Display data format for currency
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
[d,sec] = getdata(c,s,f)

d =
    LAST_PRICE: 33.34
          OPEN: 33.60

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the name of the security in `sec`.

Close the Bloomberg Server connection.

```
close(c)
```

Connect to Bloomberg Server with Port Number

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.
- The port number of the machine running the Bloomberg Server is `5678`.

```
uuid = 12345678;
ipaddress = '111.11.11.111';
port = 5678;

c = bloombergServer(uuid,ipaddress,port)

c =

  bloombergServer with properties:

               Uuid: 12345678
               User: []
             Userip: '111.11.11.112'
            Session: [1×1 BLPSession]
          IPAddress: '111.11.11.111'
               Port: 5678
            TimeOut: 0
        DatetimeType: ''
    DataReturnFormat: ''
```

The `bloombergServer` function connects to the machine running the Bloomberg Server using the port number `8194` and creates the `bloombergServer` object `c` with these properties:

- Bloomberg user identity UUID
- Bloomberg user identity object
- IP address of the machine running MATLAB

- Bloomberg V3 API Session object
- IP address of the machine running the Bloomberg Server
- Port number of the machine running the Bloomberg Server
- Number in milliseconds specifying how long MATLAB attempts to connect to the machine running the Bloomberg Server before timing out
- Date and time data type
- Data return format

Request the last and open prices for Microsoft.

```
format bank  % Display data format for currency
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
[d,sec] = getdata(c,s,f)

d =
    LAST_PRICE: 33.34
          OPEN: 33.60

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the name of the security in `sec`.

Close the Bloomberg Server connection.

```
close(c)
```

Connect to Bloomberg Server with Timeout

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.
- The port number of the machine running the Bloomberg Server is your default port number.
- The timeout value is 10 milliseconds.

```
uuid = 12345678;
ipaddress = '111.11.11.111';
port = [];
timeout = 10;

c = bloombergServer(uuid,ipaddress,port,timeout)

c =

  bloombergServer with properties:

               Uuid: 12345678
               User: []
```

```
              Userip: '111.11.11.112'
             Session: [1×1 BLPSession]
           IPAddress: '111.11.11.111'
                Port: 8194
             TimeOut: 10
        DatetimeType: ''
    DataReturnFormat: ''
```

The `bloombergServer` function connects to the machine running the Bloomberg Server using the default port number `8194` and a timeout value of 10 milliseconds. The `bloombergServer` function creates the `bloombergServer` object `c` with these properties:

- Bloomberg user identity UUID
- Bloomberg user identity object
- IP address of the machine running MATLAB
- Bloomberg V3 API Session object
- IP address of the machine running the Bloomberg Server
- Port number of the machine running the Bloomberg Server
- Number in milliseconds specifying how long MATLAB attempts to connect to the machine running the Bloomberg Server before timing out
- Date and time data type
- Data return format

Request the last and open prices for Microsoft.

```
format bank  % Display data format for currency
s = 'MSFT US Equity';
f = {'LAST_PRICE';'OPEN'};
[d,sec] = getdata(c,s,f)
```

```
d =
    LAST_PRICE: 33.34
          OPEN: 33.60

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the name of the security in `sec`.

Close the Bloomberg Server connection.

```
close(c)
```

Version History

Introduced in R2021a

See Also

Topics

“Data Server Connection Requirements” on page 1-3

“Comparing Bloomberg Connections” on page 2-4
“Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface” on page 5-39
“Retrieve Bloomberg Historical Data Using Bloomberg Server C++ Interface” on page 5-41

category

Field category search for Bloomberg Server connection V3

Syntax

```
d = category(c,f)
```

Description

`d = category(c,f)` returns category information given the search term `f` using the Bloomberg Server C++ interface.

Examples

Search for Bloomberg Last Price Field

Create a Bloomberg connection, and then request the category description of the last price field.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return data as a table by setting the `DataReturnFormat` property of the `bloomberg` object. If you do not set this property, the `category` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Request the Bloomberg category description of the last price field.

```
f = 'LAST_PRICE';
d = category(c,f);
```

Display the first three rows of the Bloomberg category description data in `d`.

```
d(1:3,:)
```

```
ans =

  3×5 table

    CATEGORY       ID                  MNEMONIC                          DESCRIPTION
```

```
    ____________    _______    ____________________________________    ____________________________________

    'Analysis'      'OP179'    'THETA_LAST'                           'Theta Last Price'
    'Analysis'      'VM048'    'DDMX_PERCENT_CHANGE_LAST_PRICE'       'DDMX Percent Change Last Price
    'Analysis'      'YL005'    'YLD_CNV_LAST'                         'Last Yield To Convention'
```

The columns in `d` are:

- Category
- Field identifier
- Field mnemonic
- Field name
- Field data type

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

f — Search term

character vector | string scalar

Search term, specified as a character vector or string scalar to denote Bloomberg fields.

Data Types: `char` | `string`

Output Arguments

d — Category data

cell array (default) | structure | table

Category data, returned as an N-by-5 cell array, a structure, or a table.

The columns (or fields) of the data types are:

- Category
- Field identifier
- Field mnemonic
- Field name
- Field data type

The data type of the category data depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2021a

See Also

`bloombergServer` | `close` | `getdata` | `history` | `realtime` | `timeseries` | `fieldinfo` | `fieldsearch`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface” on page 5-39

close

Close Bloomberg Server connection V3

Syntax

```
close(c)
```

Description

`close(c)` closes the Bloomberg Server connection V3 `c` using the Bloomberg Server C++ interface.

Examples

Close Bloomberg Connection

First, create a Bloomberg Server connection. Then, request last and open prices for a security and close the connection. The current data you see when running this code can differ from the output data here.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Request last and open prices for Microsoft.

```
[d,sec] = getdata(c,'MSFT US Equity',{'LAST_PRICE';'OPEN'})
```

```
d =
    LAST_PRICE: 33.3401
          OPEN: 33.6000

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the security in `sec`.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

Version History

Introduced in R2021a

See Also

`bloombergServer` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface” on page 5-39
“Retrieve Bloomberg Historical Data Using Bloomberg Server C++ Interface” on page 5-41
“Retrieve Bloomberg Intraday Tick Data Using Bloomberg Server C++ Interface” on page 5-45
“Retrieve Bloomberg Real-Time Data Using Bloomberg Server C++ Interface” on page 5-47

eqs

Equity screening data for Bloomberg Server connection V3

Syntax

```
d = eqs(c,sname)
d = eqs(c,sname,stype)
d = eqs(c,sname,stype,languageid)
d = eqs(c,sname,stype,languageid,group)
d = eqs(c,sname,stype,languageid,group,'OverrideFields',ov)
```

Description

`d = eqs(c,sname)` returns equity screening data given the Bloomberg Server V3 session screen name `sname`.

`d = eqs(c,sname,stype)` also specifies the screen type `stype`.

`d = eqs(c,sname,stype,languageid)` also specifies the language identifier `languageid`.

`d = eqs(c,sname,stype,languageid,group)` also specifies the optional group identifier `group`.

`d = eqs(c,sname,stype,languageid,group,'OverrideFields',ov)` also specifies the Bloomberg override fields and values `ov`.

Examples

Retrieve Equity Screening Data for Screen

Create a Bloomberg connection, and then retrieve frontier market stock data.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return data as a table by setting the `DataReturnFormat` property of the `bloomberg` object. If you do not set this property, the `eqs` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Retrieve equity screening data for the screen named `Frontier Market Stocks with 1 billion USD Market Caps`.

```
sname = 'Frontier Market Stocks with 1 billion USD Market Caps';
d = eqs(c,sname);
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  3×8 table

      Cntry                 Name                    IndGroup    MarketCap     Price_D_1     P_B
    ___________    ___________________________    ________    __________    _________    _____

    'Venezuela'    'MERCANTIL SERVICIOS FINAN-A'    'Banks'     7.3424e+12      70088      278.25
    'Venezuela'    'BANCO DEL CARIBE-A'             'Banks'     2.0442e+12      24531      2321.8
    'Venezuela'    'BANCO PROVINCIAL'               'Banks'     1.2632e+12      11715      52.344
```

The columns in d are:

- Country name
- Company name
- Industry name
- Market capitalization
- Price
- Price-to-book ratio
- Price-earnings ratio
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Retrieve Equity Screening Data for a Screen Type

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve equity screening data for the screen called `Vehicle-Engine-Parts` and the screen type equal to `'GLOBAL'`.

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL');
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    'Ticker'        'Short Name'        'Market Cap'        'Price:D-1'    'P/E'
    'HON     US'    'HONEYWELL INTL'    [69451382784.00]    [    88.51]    [16.81]
    'CMI     US'    'CUMMINS INC'       [24799526912.00]    [   132.36]    [17.28]

  Columns 6 through 8

    'Total Return YTD'    'Revenue T12M'      'EPS T12M'
    [           42.43]    [38248998912.00]    [    4.11]
    [           24.43]    [17004999936.00]    [    7.57]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen. The first row contains column headers. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Retrieve Equity Screening Data for a Screen in German

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve equity screening data for the screen called `Vehicle-Engine-Parts`, the screen type equal to `'GLOBAL'`, and return data in German.

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL','GERMAN');
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
  Columns 1 through 5

    'Ticker'        'Kurzname'          'Marktkapitalisie...'    'Preis:D-1'    'KGV'
    'HON     US'    'HONEYWELL INTL'    [     69451382784.00]    [    88.51]    [16.81]
    'CMI     US'    'CUMMINS INC'       [     24799526912.00]    [   132.36]    [17.28]

  Columns 6 through 8

    'Gesamtertrag YTD'    'Erlös T12M'        'EPS T12M'
    [           42.43]    [38248998912.00]    [    4.11]
    [           24.43]    [17004999936.00]    [    7.57]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen. The first row contains column headers in German. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Retrieve Equity Screening Data for a Screen with a Specified Screen Folder Name

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve equity screening data for the Bloomberg screen called `Vehicle-Engine-Parts`, using the Bloomberg screen type `'GLOBAL'` and the language `'ENGLISH'`, and the Bloomberg screen folder name `'GENERAL'`.

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL','ENGLISH','GENERAL');
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    'Ticker'        'Short Name'        'Market Cap'        'Price:D-1'    'P/E'
    'HON     US'    'HONEYWELL INTL'    [69451382784.00]    [    88.51]    [16.81]
    'CMI     US'    'CUMMINS INC'       [24799526912.00]    [   132.36]    [17.28]

  Columns 6 through 8

    'Total Return YTD'    'Revenue T12M'      'EPS T12M'
    [           42.43]    [38248998912.00]    [    4.11]
    [           24.43]    [17004999936.00]    [    7.57]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen. The first row contains column headers. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Retrieve Equity Screening Data Using Override Fields

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve equity screening data as of a specified date using these input arguments. The override field `PiTDate` is equivalent to the flag `AsOf` in the Bloomberg Excel Add-In.

- Bloomberg connection `c`
- Bloomberg screen is `Vehicle-Engine-Parts`
- Bloomberg screen type is `'GLOBAL'`

- Language is `'ENGLISH'`
- Bloomberg screen folder name is `'GENERAL'`
- Override field `PiTDate` is September 9, 2014

```
d = eqs(c,'Vehicle-Engine-Parts','GLOBAL','ENGLISH','GENERAL',...
    'OverrideFields',{'PiTDate','20140909'});
```

Display the first three rows in the returned data `d`.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    'Ticker'        'Short Name'        'Market Cap'    'Price:D-1'       'P/E'
    'HON US'     'HONEYWELL INTL'      [7.3919e+10]    [  94.4600]    [17.8087]
    'TSLA US'    'TESLA MOTORS'        [3.4707e+10]    [ 278.4800]    [    NaN]

  Columns 6 through 8

    'Total Return YTD'    'Revenue T12M'    'EPS T12M'
    [          4.8907]    [  3.9966e+10]    [  5.1600]
    [         85.1239]    [  2.4365e+09]    [ -1.3500]
```

`d` contains Bloomberg equity screening data for the `Vehicle-Engine-Parts` screen as of September 9, 2014. The first row contains column headers. The subsequent rows contain the returned data. The columns in `d` are:

- Ticker symbol
- Company name
- Market capitalization
- Price
- Price-earnings ratio
- Total return year-to-date
- Revenue
- Earnings per share

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

`sname` — Screen name

character vector | string scalar

Screen name, specified as a character vector or string scalar to denote the Bloomberg V3 session screen name to execute. The screen can be a customized equity screen or one of the Bloomberg example screens accessed by using the **EQS <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

stype — Screen type
'GLOBAL' | 'PRIVATE'

Screen type, specified as one of the two preceding values to denote the Bloomberg screen type. 'GLOBAL' denotes a Bloomberg screen name and 'PRIVATE' denotes a customized screen name. When using the optional group input argument, stype cannot be set to 'PRIVATE' for customized screen names.

languageid — Language identifier
character vector | string scalar

Language identifier, specified as a character vector or string to denote the language for the returned data. This argument is optional.

Data Types: char | string

group — Group identifier
character vector | string scalar

Group identifier, specified as a character vector or string to denote the Bloomberg screen folder name accessed by using the **EQS <GO>** option from the Bloomberg terminal. This argument is optional. When using this argument, stype cannot be set to 'PRIVATE' for customized screen names.

Data Types: char | string

ov — Bloomberg override field values
cell array

Bloomberg override field values, specified as an n-by-2 cell array. The first column of the cell array is the override field. The second column is the override value.

Example: {'PiTDate','20140909'}

Data Types: cell

Output Arguments

d — Equity screening data
cell array (default) | structure | table

Equity screening data, returned as a cell array, structure, or table. The data type of the equity screening data depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2021a

See Also

bloombergServer | close | getdata | tahistory

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface” on page 5-39

fieldinfo

Field information for Bloomberg Server connection V3

Syntax

```
d = fieldinfo(c,f)
```

Description

`d = fieldinfo(c,f)` returns field information using the `bloombergServer` object `c` with the Bloomberg Server C++ interface and field mnemonic `f`.

Examples

Retrieve Information for Last Price Field

Create a Bloomberg connection, and then retrieve information for the last price field.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return data as a table by setting the `DataReturnFormat` property of the `bloomberg` object. If you do not set this property, the `fieldinfo` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Retrieve the Bloomberg field information for the `LAST_PRICE` field.

```
f = 'LAST_PRICE';
d = fieldinfo(c,f);
```

Display the last four columns in the returned Bloomberg information.

```
d(:,2:5)
```

```
ans =

  1×4 table

      ID          MNEMONIC             DESCRIPTION          DATATYPE
```

```
    _______    _____________    ________________________    ________

    'RQ005'    'LAST_PRICE'    'Last Trade/Last Price'    'Double'
```

The columns in `d` are:

- Field identifier
- Field mnemonic
- Field name
- Field data type

You can also access the Bloomberg help information in the first column.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

`f` — Field mnemonic

character vector | string scalar

Field mnemonic, specified as a character vector or string scalar that denotes the Bloomberg field information to retrieve.

Data Types: `char` | `string`

Output Arguments

`d` — Field information

cell array (default) | structure | table

Field information, returned as an N-by-5 cell array, a structure, or a table.

The columns (or fields) of the data types are:

- Field help
- Field identifier
- Field mnemonic
- Field name
- Field data type

The data type of the field information depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2021a

See Also

`bloombergServer` | `close` | `getdata` | `history` | `realtime` | `timeseries` | `category` | `fieldsearch`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface” on page 5-39

fieldsearch

Field search for Bloomberg Server connection V3

Syntax

```
d = fieldsearch(c,f)
```

Description

`d = fieldsearch(c,f)` returns field information using the `bloombergServer` object `c` with the Bloomberg Server C++ interface and search term `f`.

Examples

Retrieve Information for Last Price Field

Create a Bloomberg connection, and then retrieve information for the last price field.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return data as a table by setting the `DataReturnFormat` property of the `bloombergServer` object. If you do not set this property, the `fieldsearch` function returns data as a cell array.

```
c.DataReturnFormat = 'table';
```

Return information for the search term `LAST_PRICE`.

```
f = 'LAST_PRICE';
d = fieldsearch(c,f);
```

Display the first three rows of the field information in `d`.

```
d(1:3,:)
```

```
ans =

  3×5 table

           CATEGORY             ID            MNEMONIC                    DESCRIPTION
```

```
_________________________    _______    ____________________    ___________________________________

'Market Activity/Last'    'PR005'    'PX_LAST'               'Last Price'
'Market Activity/Last'    'RQ005'    'LAST_PRICE'            'Last Trade/Last Price'
'Market Activity/Last'    'PR910'    'CRNCY_ADJ_PX_LAST'     'Currency Adjusted Last Price'
```

The columns in `d` are:

- Category
- Field identifier
- Field mnemonic
- Field name
- Field data type

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

`f` — Search term

character vector | string scalar

Search term, specified as a character vector or string scalar that denotes the Bloomberg field descriptive data to retrieve.

Data Types: `char` | `string`

Output Arguments

`d` — Field data

cell array (default) | structure | table

Field data, returned as an N-by-5 cell array, a structure, or a table.

The columns (or fields) of the data types are:

- Category
- Field identifier
- Field mnemonic
- Field name
- Field data type

The data type of the field data depends on the DataReturnFormat property of the connection object.

Version History

Introduced in R2021a

See Also

`bloombergServer` | `close` | `getdata` | `history` | `realtime` | `timeseries` | `category` | `fieldinfo`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface” on page 5-39

get

Properties of Bloomberg Server connection V3

Syntax

```
v = get(c)
v = get(c,properties)
```

Description

`v = get(c)` returns a structure where each field name is the name of a property of the `bloombergServer` object `c`, which uses the Bloomberg Server C++ interface, and each field contains the value of that property.

`v = get(c,properties)` returns the value of the specified properties `properties` for the Bloomberg V3 connection object.

Examples

Retrieve Bloomberg Connection Properties

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the Bloomberg connection properties.

```
v = get(c)
```

```
v =

  struct with fields:

      session: [1×1 datafeed.internal.BLPSession]
    ipaddress: "localhost"
         port: 8194.00
```

`v` is a structure containing the Bloomberg session object, IP address, port number, and timeout value.

Close the Bloomberg connection.

```
close(c)
```

Retrieve One Bloomberg Connection Property

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the port number from the Bloomberg connection object by specifying `'port'` as a character vector.

```
property = "port";
v = get(c,property)
```

```
v =

        8194
```

`v` is a double that contains the port number of the Bloomberg connection object.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Two Bloomberg Connection Properties

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Create a cell array `properties` with character vectors `'session'` and `'port'`. Retrieve the Bloomberg session object and port number from the Bloomberg connection object.

```
properties = ["session" "port"];
v = get(c,properties)
```

```
v =

  struct with fields:

    session: [1x1 com.bloomberglp.blpapi.Session]
       port: 8194
```

`v` is a structure containing the Bloomberg session object and port number.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

`properties` — Property names

character vector | string scalar | cell array of character vectors | string array

Property names, specified as a character vector, string scalar, cell array of character vectors, or string array containing Bloomberg connection property names. The property names are `session`, `ipaddress`, `port`, and `timeout`.

Data Types: `char` | `cell` | `string`

Output Arguments

`v` — Bloomberg connection properties

numeric scalar | character vector | object | structure

Bloomberg connection properties, returned as these data types depending on the requested properties.

Requested Properties	Data Type
Port number or timeout	Numeric scalar
IP address	Character vector
Bloomberg session	Object
All properties	Structure

Version History

Introduced in R2021a

See Also

`bloombergServer` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface” on page 5-39

getbulkdata

Bulk data with header information for Bloomberg Server connection V3

Syntax

```
d = getbulkdata(c,s,f)
d = getbulkdata(c,s,f,o,ov)
d = getbulkdata(c,s,f,o,ov,Name,Value)
[d,sec] = getbulkdata( ___ )
```

Description

`d = getbulkdata(c,s,f)` returns the bulk data for the fields `f` for the security list `s` using the `bloombergServer` object `c` with the Bloomberg Server C++ interface.

`d = getbulkdata(c,s,f,o,ov)` returns the bulk data using the override fields `o` with corresponding override values `ov`.

`d = getbulkdata(c,s,f,o,ov,Name,Value)` returns the bulk data with additional options specified by one or more name-value pair arguments for Bloomberg request settings.

`[d,sec] = getbulkdata( ___ )` additionally returns the security list `sec` using any of the input argument combinations in the previous syntaxes.

Examples

Return Specific Field for Given Security

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return the dividend history for IBM.

```
security = 'IBM US Equity';
field = 'DVD_HIST'; % Dividend history field

[d,sec] = getbulkdata(c,security,field)
```

```
d =
```

```
    DVD_HIST: {{149x7 cell}}

sec =

    'IBM US Equity'
```

`d` is a structure with one field that contains a cell array with the returned bulk data. `sec` contains the IBM security name.

Display the dividend history with the associated header information by accessing the structure field `DVD_HIST`. This field is a cell array that contains one cell array. The nested cell array contains the dividend history data. Access the contents of the nested cell using cell array indexing.

```
d.DVD_HIST{1}
```

```
ans =

  Columns 1 through 6

    'Declared Date'    'Ex-Date'    'Record Date'    'Payable Date'    'Dividend Amount'    'Dividend Frequency'
    [       735536]    [ 735544]    [     735546]    [      735578]    [         0.95]      'Quarter'
    [       735445]    [ 735453]    [     735455]    [      735487]    [         0.95]      'Quarter'
    [       735354]    [ 735362]    [     735364]    [      735395]    [         0.95]      'Quarter'
    ...

  Column 7

    'Dividend Type'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
    ...
```

The first row of the dividend history data is the header information that describes the contents of each column.

Close the Bloomberg connection.

```
close(c)
```

Return Specific Field Using Override Values

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return the dividend history for IBM with dividend dates from January 1, 2004, through January 1, 2005.

```
security = 'IBM US Equity';
field = 'DVD_HIST';                         % Dividend history field
override = {'DVD_START_DT','DVD_END_DT'};  % Dividend start and
```

```
                                                % End dates
overridevalues = {'20040101','20050101'};

[d,sec] = getbulkdata(c,security,field,override,overridevalues)

d =

    DVD_HIST: {{5x7 cell}}

sec =

    'IBM US Equity'
```

`d` is a structure with one field that contains a cell array with the returned bulk data. `sec` contains the IBM security name.

Display the dividend history with the associated header information by accessing the structure field `DVD_HIST`. This field is a cell array that contains one cell array. The nested cell array contains the dividend history data. Access the contents of the nested cell using cell array indexing.

```
d.DVD_HIST{1}
```

```
ans =

  Columns 1 through 6

    'Declared Date'    'Ex-Date'    'Record Date'    'Payable Date'    'Dividend Amount'    'Dividend Frequency'
    [       732246]    [ 732259]    [     732261]    [      732291]    [         0.18]      'Quarter'
    [       732155]    [ 732165]    [     732169]    [      732200]    [         0.18]      'Quarter'
    [       732064]    [ 732073]    [     732077]    [      732108]    [         0.18]      'Quarter'
    [       731973]    [ 731983]    [     731987]    [      732016]    [         0.16]      'Quarter'

  Column 7

    'Dividend Type'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
```

The first row of the dividend history data is the header information that describes the contents of each column.

Close the Bloomberg connection.

```
close(c)
```

Return Specific Field Using Name-Value Pair Arguments

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return the closing price and dividend history for IBM with dividend dates from January 1, 2004, through January 1, 2005. Specify the data return format as a character vector by setting the name-value pair argument `'returnFormattedValue'` to `'true'`.

```
security = 'IBM US Equity';
fields = {'LAST_PRICE','DVD_HIST'};        % Closing price and
                                           % Dividend history fields
override = {'DVD_START_DT','DVD_END_DT'};  % Dividend start and
                                           % End dates
overridevalues = {'20040101','20050101'};

[d,sec] = getbulkdata(c,security,fields,override,overridevalues,...
                      'returnFormattedValue',true)
```

```
d =

      DVD_HIST: {{5x7 cell}}
    LAST_PRICE: {'188.74'}

sec =

    'IBM US Equity'
```

`d` is a structure with two fields. The first field `DVD_HIST` contains a cell array with the dividend historical data as a cell array. The second field `LAST_PRICE` contains a cell array with the closing price as a character vector. `sec` contains the IBM security name.

Display the closing price.

```
d.LAST_PRICE
```

```
ans =

    '188.74'
```

Display the dividend history with the associated header information by accessing the structure field `DVD_HIST`. This field is a cell array that contains one cell array. The nested cell array contains the dividend history data. Access the contents of the nested cell using cell array indexing.

```
d.DVD_HIST{1}
```

```
ans =

  Columns 1 through 6

    'Declared Date'    'Ex-Date'    'Record Date'    'Payable Date'    'Dividend Amount'    'Dividend Frequency'
    [       732246]    [ 732259]    [     732261]    [      732291]    [         0.18]      'Quarter'
    [       732155]    [ 732165]    [     732169]    [      732200]    [         0.18]      'Quarter'
    [       732064]    [ 732073]    [     732077]    [      732108]    [         0.18]      'Quarter'
    [       731973]    [ 731983]    [     731987]    [      732016]    [         0.16]      'Quarter'

  Column 7

    'Dividend Type'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
    'Regular Cash'
```

The first row of the dividend history data is the header information that describes the contents of each column.

Close the Bloomberg connection.

```
close(c)
```

Return Bulk Data as Table with Datetime

Create a Bloomberg connection, and then request dividend history data. The `getbulkdata` function returns data for dates as a `datetime` array.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `getbulkdata` function returns data as a structure.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Return the dividend history for IBM.

```
s = 'IBM US Equity';
f = 'DVD_HIST'; % Dividend history field

d = getbulkdata(c,s,f);
```

Display the first three rows of the table.

```
d.DVD_HIST{1}(1:3,:)
```

```
ans =

  3×7 table

        DeclaredDate             ExmDate             RecordDate             PayableDate
    ____________________    ____________________    ____________________    ____________

    31-Oct-2017 00:00:00    09-Nov-2017 00:00:00    10-Nov-2017 00:00:00    09-Dec-2017 00:00:00
    25-Jul-2017 00:00:00    08-Aug-2017 00:00:00    10-Aug-2017 00:00:00    09-Sep-2017 00:00:00
    25-Apr-2017 00:00:00    08-May-2017 00:00:00    10-May-2017 00:00:00    10-Jun-2017 00:00:00
```

Display three declared dates. The `DeclaredDate` variable is a `datetime` array.

```
d.DVD_HIST{1}.DeclaredDate(1:3)
```

```
ans =

  3×1 datetime array

   31-Oct-2017 00:00:00
   25-Jul-2017 00:00:00
   25-Apr-2017 00:00:00
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

f — Bloomberg data fields

character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

o — Bloomberg override field

`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field name. A cell array of character vectors or string array denotes multiple Bloomberg override field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `'END_DT'`

Data Types: `char` | `cell` | `string`

ov — Bloomberg override field value

`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field value, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field value. A cell array of character vectors or string array denotes multiple Bloomberg override field values. Use this field value to filter the Bloomberg data result set.

Example: `'20100101'`

Data Types: `char` | `cell` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'returnFormattedValue',true`

`returnEids` — Entitlement identifiers
true | false

Entitlement identifiers, specified as the comma-separated pair consisting of `'returnEids'` and a Boolean. `true` adds a name and value for the entitlement identifier (EID) date to the return data.

Data Types: `logical`

`returnFormattedValue` — Return format
true | false

Return format, specified as the comma-separated pair consisting of `'returnFormattedValue'` and a Boolean. `true` forces all data to be returned as the data type character vector.

Data Types: `logical`

`useUTCTime` — Date time format
true | false

Date time format, specified as the comma-separated pair consisting of `'useUTCTime'` and a Boolean. `true` returns date and time values as Coordinated Universal Time (UTC) and `false` defaults to the Bloomberg **TZDF <GO>** settings of the requestor.

Data Types: `logical`

`forcedDelay` — Latest reference data
true | false

Latest reference data, specified as the comma-separated pair consisting of `'forcedDelay'` and a Boolean. `true` returns the latest data up to the delay period specified by the exchange for the security.

Data Types: `logical`

Output Arguments

`d` — Bloomberg data
structure (default) | table | timetable

Bloomberg data, returned as a structure, table, or timetable. The data type of the Bloomberg data depends on the DataReturnFormat and DatetimeType properties of the connection object. For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`sec` — Security list
cell array of character vectors

Security list, returned as a cell array of character vectors for the corresponding securities in `s`. The contents of `sec` are identical in value and order to `s`. You can return securities with any of the following identifiers:

- `buid`
- `cats`
- `cins`
- `common`
- `cusip`
- `isin`
- `sedol1`
- `sedol2`
- `sicovam`
- `svm`
- `ticker` (default)
- `wpk`

Version History
Introduced in R2021a

See Also
`bloombergServer` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics
"Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface" on page 5-39

getdata

Current data for Bloomberg Server connection V3

Syntax

```
d = getdata(c,s,f)
d = getdata(c,s,f,o,ov)
d = getdata(c,s,f,o,ov,Name,Value)
[d,sec] = getdata( ___ )
```

Description

`d = getdata(c,s,f)` returns the data for the fields `f` for the security list `s` using the `bloombergServer` object with the Bloomberg Server C++ interface. `getdata` accesses the Bloomberg reference data service.

`d = getdata(c,s,f,o,ov)` returns the data using the override fields `o` with corresponding override values `ov`.

`d = getdata(c,s,f,o,ov,Name,Value)` returns the data using name-value pair arguments for additional Bloomberg request settings.

`[d,sec] = getdata( ___ )` additionally returns the security list `sec` using any of the input argument combinations in the previous syntaxes.

Examples

Last and Open Price for Security

First, create a Bloomberg Server connection. Then, request last and open prices for a security. The current data you see when running this code can differ from the output data here.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Request last and open prices for Microsoft.

```
[d,sec] = getdata(c,'MSFT US Equity',{'LAST_PRICE';'OPEN'})
```

```
d =
    LAST_PRICE: 33.3401
```

```
          OPEN: 33.6000

sec =
    'MSFT US Equity'
```

`getdata` returns a structure `d` with the last and open prices. Also, `getdata` returns the security in `sec`.

Close the connection.

```
close(c)
```

Specified Fields Given Override Fields and Values

First, create a Bloomberg Server connection. Then, request data for specific fields for a security using an override field and value. The current data you see when running this code can differ from the output data here.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Request data for Bloomberg fields `'YLD_YTM_ASK'`, `'ASK'`, and `'OAS_SPREAD_ASK'` when the Bloomberg field `'OAS_VOL_ASK'` is `'14.000000'`.

```
[d,sec] = getdata(c,'030096AF8 Corp',...
   {'YLD_YTM_ASK','ASK','OAS_SPREAD_ASK','OAS_VOL_ASK'},...
   {'OAS_VOL_ASK'},{'14.000000'})

d =
       YLD_YTM_ASK: 5.6763
               ASK: 120.7500
    OAS_SPREAD_ASK: 307.9824
       OAS_VOL_ASK: 14

sec =
    '030096AF8 Corp'
```

`getdata` returns a structure `d` with the resulting values for the requested fields.

Close the connection.

```
close(c)
```

Request for Security Using CUSIP Number

First, create a Bloomberg Server connection. Then, use the CUSIP number for a security to request last price. The current data you see when running this code can differ from the output data here.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Request the last price for IBM with the CUSIP number.

```
d = getdata(c,'/cusip/459200101','LAST_PRICE')
```

```
d =
    LAST_PRICE: 182.5100
```

`getdata` returns a structure `d` with the last price.

Close the connection.

```
close(c)
```

Last Price for Security with Pricing Source

First, create a Bloomberg Server connection. Then, request the last price for a security. Specify the security using the CUSIP number with a pricing source. The current data you see when running this code can differ from the output data here.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Specify IBM with the CUSIP number and the pricing source `BGN` after the `@` symbol.

```
d = getdata(c,'/cusip/459200101@BGN','LAST_PRICE')
```

```
d =
    LAST_PRICE: 186.81
```

`getdata` returns a structure `d` with the last price.

Close the connection.

```
close(c)
```

Constituent Weights Using Date Override

First, create a Bloomberg Server connection. Then, request the constituent weights of an index using a date override. The current data you see when running this code can differ from the output data here.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return the constituent weights for the Dow Jones Index as of January 1, 2010, using a date override with the required date format `YYYYMMDD`.

```
d = getdata(c,'DJX Index','INDX_MWEIGHT','END_DT','20100101')
```

```
d =
    INDX_MWEIGHT: {{30x2 cell}}
```

`getdata` returns a structure `d` with a cell array where the first column is the index and the second column is the constituent weight.

Display the constituent weights for each index.

```
d.INDX_MWEIGHT{1,1}
```

```
ans =
    'AA UN'      [1.1683]
    'AXP UN'     [2.9366]
    'BA UN'      [3.9229]
    'BAC UN'     [1.0914]
    ...
```

Close the connection.

```
close(c)
```

Current Data and Dates as Table with Datetime

Create a Bloomberg connection, and then request current data for specific fields. The `getdata` function returns data for dates as a `datetime` array.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `getdata` function returns data as a structure.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Request current data for these fields:

- Last update date
- Last price
- Number of trades
- Previous real-time trading date

```
s = 'IBM US Equity';
f = {'LAST_UPDATE_DT','LAST_PRICE', ...
    'NUM_TRADES_RT','PREV_TRADING_DT_REALTIME'};
d = getdata(c,s,f)
```

```
d =

  1×4 table

      LAST_UPDATE_DT       LAST_PRICE    NUM_TRADES_RT    PREV_TRADING_DT_REALTIME
    ___________________    __________    _____________    ________________________

    21-Dec-2017 00:00:00      152.2           24846          20-Dec-2017 00:00:00
```

Display the last update date. This date is a `datetime` array.

```
d.LAST_UPDATE_DT
```

```
ans =
```

```
  datetime

   21-Dec-2017 00:00:00
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

f — Bloomberg data fields

character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

o — Bloomberg override field

`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field name. A cell array of character vectors or string array denotes multiple Bloomberg override field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `'END_DT'`

Data Types: `char` | `cell` | `string`

ov — Bloomberg override field value

`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field value, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field value. A

cell array of character vectors or string array denotes multiple Bloomberg override field values. Use this field value to filter the Bloomberg data result set.

Example: `'20100101'`

Data Types: `char` | `cell` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'returnEids',true`

`returnEids` — Entitlement identifiers
true | false

Entitlement identifiers, specified as the comma-separated pair consisting of `'returnEids'` and a Boolean. `true` adds a name and value for the entitlement identifier (EID) date to the return data.

Data Types: `logical`

`returnFormattedValue` — Return format
true | false

Return format, specified as the comma-separated pair consisting of `'returnFormattedValue'` and a Boolean. `true` forces all data to be returned as the data type character vector.

Data Types: `logical`

`useUTCTime` — Date time format
true | false

Date time format, specified as the comma-separated pair consisting of `'useUTCTime'` and a Boolean. `true` returns date and time values as Coordinated Universal Time (UTC) and `false` defaults to the Bloomberg **TZDF <GO>** settings of the requestor.

Data Types: `logical`

`forcedDelay` — Latest reference data
true | false

Latest reference data, specified as the comma-separated pair consisting of `'forcedDelay'` and a Boolean. `true` returns the latest data up to the delay period specified by the exchange for the security.

Data Types: `logical`

Output Arguments

`d` — Bloomberg data
structure (default) | table | timetable

Bloomberg data, returned as a structure, table, or timetable. The data type of the Bloomberg data depends on the DataReturnFormat and DatetimeType properties of the connection object. For details

about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`sec` — Security list
cell array of character vectors

Security list, returned as a cell array of character vectors for the corresponding securities in `s`. The contents of `sec` are identical in value and order to `s`. You can return securities with any of the following identifiers:

- `buid`
- `cats`
- `cins`
- `common`
- `cusip`
- `isin`
- `sedol1`
- `sedol2`
- `sicovam`
- `svm`
- `ticker` (default)
- `wpk`

Tips

- Bloomberg V3 data supports additional name-value pair arguments. To access further information on these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.
- You can check data and field availability by using the Bloomberg Excel Add-In.

Version History

Introduced in R2021a

See Also

`bloombergServer` | `close` | `history` | `realtime` | `timeseries`

Topics

"Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface" on page 5-39

history

Historical data for Bloomberg Server connection V3

Syntax

```
d = history(c,s,f,fromdate,todate)
d = history(c,s,f,fromdate,todate,period)
d = history(c,s,f,fromdate,todate,period,currency)
d = history(c,s,f,fromdate,todate,period,currency,Name,Value)
[d,sec] = history( ___ )
```

Description

`d = history(c,s,f,fromdate,todate)` returns the historical data for the security list `s` for the fields `f` for the dates `fromdate` through `todate` using the `bloombergServer` object `c` with the Bloomberg Server C++ interface. Date strings can be input in any format recognized by MATLAB. `sec` is the security list that maps the order of the return data. The return data `d` is sorted to match the input order of `s`.

`d = history(c,s,f,fromdate,todate,period)` returns the historical data for the fields `f` and the dates `fromdate` through `todate` with a specific periodicity `period`.

`d = history(c,s,f,fromdate,todate,period,currency)` returns the historical data for the security list `s` for the fields `f` and the dates `fromdate` through `todate` based on the given currency `currency`.

`d = history(c,s,f,fromdate,todate,period,currency,Name,Value)` returns the historical data for the security list `s` using additional options specified by one or more name-value pair arguments.

`[d,sec] = history( ___ )` additionally returns the security list `sec` using any of the input argument combinations in the previous syntaxes. The return data, `d` and `sec`, are sorted to match the input order of `s`.

Examples

Daily Closing Prices Within Date Range

First, create a Bloomberg Desktop connection. Then, retrieve the daily closing price for a security within a date range.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Get the daily closing price from August 1, 2010, through August 10, 2010, for the IBM security.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','8/10/2010')

d =

     734352.00        123.55
     734353.00        123.18
     734354.00        124.03
     734355.00        124.56
     734356.00        123.58
     734359.00        125.34
     734360.00        125.19


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Monthly Closing Prices Within Date Range

First, create a Bloomberg Desktop connection. Then, retrieve the monthly closing prices for a security within a date range.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Get the monthly closing price from August 1, 2010, through December 10, 2010, for the IBM security.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','12/10/2010','monthly')

d =
```

```
     734360.00        125.19
     734391.00        121.53
     734421.00        131.85
     734452.00        139.78
     734482.00        138.13


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Monthly Closing Prices Within Date Range Using US Currency

First, create a Bloomberg Desktop connection. Then, retrieve the monthly closing prices for a security within a date range. Specify prices using the US currency.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Get the monthly closing price from August 1, 2010, through December 10, 2010, for the IBM security in US currency `'USD'`.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','12/10/2010','monthly','USD')

d =

     734360.00        125.19
     734391.00        121.53
     734421.00        131.85
     734452.00        139.78
     734482.00        138.13


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Monthly Closing Prices Within Date Range Using Currency with Specified Period

First, create a Bloomberg Desktop connection. Then, retrieve the monthly closing prices for a security within a date range. Specify prices using the US currency. Specify period values to customize the returned data.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Get the monthly closing price from August 1, 2010, through August 1, 2011, for the IBM security in US currency. The period values `'monthly'`, `'actual'`, and `'all_calendar_days'` specify returning actual monthly data for all calendar days. The period value `'nil_value'` specifies filling missing data values with a `NaN`.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','8/01/2011',{'monthly','actual',...
                  'all_calendar_days','nil_value'},'USD')
```

```
d =

     734351.00        128.40
     734382.00        125.77
     734412.00        135.64
     734443.00        143.32
     734473.00        144.41
     734504.00        146.76
     734535.00        163.56
     734563.00        159.97
     734594.00        164.27
     734624.00        170.58
     734655.00        166.56
     734685.00        174.54
     734716.00        180.75


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Daily Closing Prices Within Date Range Using Currency with Name-Value Pairs

First, create a Bloomberg Desktop connection. Then, retrieve the daily closing prices for a security within a date range. Specify prices using the US currency. Use name-value pair arguments to adjust the prices.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Get the daily closing price from August 1, 2010, through August 10, 2010, for the IBM security in U.S. currency `'USD'`. The prices are adjusted for normal cash and splits.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE',...
                  '8/01/2010','8/10/2010','daily','USD',...
                  'adjustmentNormal',true,...
                  'adjustmentSplit',true)
```

```
d =

     734352.00        123.55
     734353.00        123.18
     734354.00        124.03
     734355.00        124.56
     734356.00        123.58
     734359.00        125.34
     734360.00        125.19


sec =

    'IBM US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Daily Closing Prices Using CUSIP Number and Pricing Source

First, create a Bloomberg Desktop connection. Then, retrieve the daily closing prices for a security within a date range. Specify the security using the CUSIP number and a pricing source.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Get the daily closing price from January 1, 2012, through January 1, 2013, for the security specified with a CUSIP number `/cusip/459200101` and with pricing source `BGN`.

`d` contains the numeric representation for the date in the first column and the closing price in the second column.

```
d = history(c,'/cusip/459200101@BGN','LAST_PRICE',...
            '01/01/2012','01/01/2013')

d =

     734871.00        180.69
     734872.00        179.96
     734873.00        179.10
     ...
```

Close the Bloomberg connection.

```
close(c)
```

Closing Prices Within Date Range Using International Date Format

First, create a Bloomberg Desktop connection. Then, retrieve the closing prices for a security within a date range. Specify the dates for the range using an international date format.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return the closing price for the given dates in international format for the security `'MSFT@BGN US Equity'`.

```
stDt = datenum('01/06/11','dd/mm/yyyy');
endDt = datenum('01/06/12','dd/mm/yyyy');
[d,sec] = history(c,'MSFT@BGN US Equity','LAST_PRICE',...
                  stDt,endDt,{'previous_value','all_calendar_days'})

d =

     734655.00         22.92
     734656.00         22.72
     734657.00         22.42
     ...

sec =

    'MSFT@BGN US Equity'
```

`d` contains the numeric representation for the date in the first column and the closing price in the second column. `sec` contains the name of the IBM security.

Close the Bloomberg connection.

```
close(c)
```

Median Estimated Earnings Per Share Using Override Fields

First, create a Bloomberg Desktop connection. Then, retrieve the median earnings per share for a security within a date range. Specify an override field and value.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the median estimated earnings per share for AkzoNobel from October 1, 2010, through October 30, 2010. When specifying Bloomberg override fields, use the character vector `'overrideFields'`. The `overrideFields` argument must be an `n`-by-`2` cell array, where the first column is the override field and the second column is the override value.

```
d = history(c,'AKZA NA Equity', ...
            'BEST_EPS_MEDIAN',datenum('01.10.2010', ...
```

```
            'dd.mm.yyyy'),datenum('30.10.2010','dd.mm.yyyy'), ...
            {'daily','calendar'},[],'overrideFields', ...
            {'BEST_FPERIOD_OVERRIDE','BF'})

d =

     734412.00          3.75
     734415.00          3.75
     734416.00          3.75
     ...
```

`d` returns the numeric representation for the date in the first column and the median estimated earnings per share in the second column.

Close the Bloomberg connection.

```
close(c)
```

Historical Data as Table with Dates

Create a Bloomberg connection, and then retrieve closing prices for a historical date range. The `history` function returns data for dates as a `datetime` array.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `history` function returns data as a numeric array.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve historical closing prices for IBM from August 1, 2010, through August 10, 2010. `d` is a table that contains dates as a `datetime` array.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE', ...
    '8/01/2010','8/10/2010')


d =
```

```
  7×2 table

      DATE        LAST_PRICE
    ___________    __________

    02-Aug-2010      130.76
    03-Aug-2010      130.37
    04-Aug-2010      131.27
    05-Aug-2010      131.83
    06-Aug-2010      130.14
    09-Aug-2010      132.00
    10-Aug-2010      131.84


sec =

  1×1 cell array

    {'IBM US Equity'}
```

Access dates in the returned data.

```
d.DATE
```

```
ans =

  7×1 datetime array

   02-Aug-2010
   03-Aug-2010
   04-Aug-2010
   05-Aug-2010
   06-Aug-2010
   09-Aug-2010
   10-Aug-2010
```

Close the Bloomberg connection.

```
close(c)
```

Historical Data as Timetable

Create a Bloomberg connection, and then retrieve closing prices for a historical date range. The `history` function returns data as a `timetable`.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return data as a `timetable` by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `history` function returns data as a numeric array.

```
c.DataReturnFormat = 'timetable';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve historical closing prices for IBM from August 1, 2010, through August 10, 2010. `d` is a `timetable` that contains dates in the first column.

```
[d,sec] = history(c,'IBM US Equity','LAST_PRICE', ...
    '8/01/2010','8/10/2010')
```

```
d =

  7×1 timetable

       DATE        LAST_PRICE
    ___________    __________

    02-Aug-2010      130.76
    03-Aug-2010      130.37
    04-Aug-2010      131.27
    05-Aug-2010      131.83
    06-Aug-2010      130.14
    09-Aug-2010      132.00
    10-Aug-2010      131.84


sec =

  1×1 cell array

    {'IBM US Equity'}
```

Access dates in the returned data.

```
d.DATE
```

```
ans =

  7×1 datetime array

   02-Aug-2010
   03-Aug-2010
   04-Aug-2010
   05-Aug-2010
   06-Aug-2010
   09-Aug-2010
```

```
10-Aug-2010
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

f — Bloomberg data fields

character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

period — Periodicity

`'daily'` | `'weekly'` | `'monthly'` | `'quarterly'` | ...

Periodicity, specified as one of these values to denote the data to return. For specifying multiple values, use a cell array. For example, when `period` is set to `{'daily','all_calendar_days'}`, `history` returns daily data for all calendar days, and reports missing data as `NaN`s. When `period` is set to `'active_days_only'`, `history` returns data using the default periodicity for active trading days only. The default periodicity depends on the security. If a security is reported on a monthly basis, the default periodicity is monthly. These tables show the values for `period`.

To specify the periodicity of the return data, see this table.

Value	Description
`'daily'`	Return data for each day.
`'weekly'`	Return data for each week.
`'monthly'`	Return data for each month.
`'quarterly'`	Return data for each quarter.

Value	Description
'semi_annually'	Return data semiannually.
'yearly'	Return data for each year.

The anchor date is the date to which all other reported dates are related. To specify the anchor date, see this table.

Value	Description
'actual'	Anchor date specification for an actual date. For this function, for periodicities other than daily, `todate` is the anchor date. If the period is weekly and the `todate` is a Thursday, every data point is a Thursday, or the nearest prior business day to Thursday. If the period is monthly and the `todate` is the 20th of a month, every data point is the 20th of each month in the date range.
'calendar'	Anchor date specification for a calendar year.
'fiscal'	Anchor date specification for a fiscal year.
'none'	Do not specify the anchor date.

To specify returning data for particular days, see this table.

Value	Description
'non_trading_weekdays'	Return data for all weekdays.
'all_calendar_days'	Return data for all calendar days.
'active_days_only'	Return data for only active trading days.

To specify how to fill missing values, see this table.

Value	Description
'previous_value'	Fill missing values with previous values for dates without trading activity for the security. If no previous value exists in the month before the `fromdate`, this function retains the missing values.
'nil_value'	Fill missing values with a `NaN` for dates without trading activity for the security.

Data Types: `char` | `cell`

`currency` — Currency
character vector | string scalar

Currency, specified as a character vector or string scalar to denote the ISO code for the currency of the returned data. For example, to specify output money values in U.S. currency, use `USD` for this argument.

Data Types: `char` | `string`

`fromdate` — Beginning date

double scalar | character vector | string scalar | datetime

Beginning date for the historical data, specified as a double scalar, character vector, string scalar, or `datetime` array. You can specify dates in any of the formats supported by `datestr` and `datenum` that show a year, month, and day.

Data Types: `datetime` | `double` | `char` | `string`

`todate` — End date

double scalar | character vector | string scalar | datetime

End date for the historical data, specified as a double scalar, character vector, string scalar, or `datetime` array. You can specify dates in any of the formats supported by `datestr` and `datenum` that show a year, month, and day.

Data Types: `datetime` | `double` | `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'adjustmentNormal',true`

`overrideFields` — Override fields

cell array

Override fields, specified as the comma-separated pair consisting of `'overrideFields'` and an `n`-by-`2` cell array. The first column of the cell array is the override field and the second column is the override value.

Example: `'overrideFields',
{'IVOL_DELTA_LEVEL','DELTA_LVL_10';'IVOL_DELTA_PUT_OR_CALL','IVOL_PUT';'IVOL_MATURITY','MATURITY_1STM'}`

Data Types: `cell`

`adjustmentNormal` — Historical normal pricing adjustment

true | false

Historical normal pricing adjustment, specified as the comma-separated pair consisting of `'adjustmentNormal'` and a Boolean to reflect:

- Regular Cash
- Interim
- 1st Interim
- 2nd Interim
- 3rd Interim
- 4th Interim

- 5th Interim
- Income
- Estimated
- Partnership Distribution
- Final
- Interest on Capital
- Distribution
- Prorated

For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

`adjustmentAbnormal` — Historical abnormal pricing adjustment
true | false

Historical abnormal pricing adjustment, specified as the comma-separated pair consisting of `'adjustmentAbnormal'` and a Boolean to reflect:

- Special Cash
- Liquidation
- Capital Gains
- Long-Term Capital Gains
- Short-Term Capital Gains
- Memorial
- Return of Capital
- Rights Redemption
- Miscellaneous
- Return Premium
- Preferred Rights Redemption
- Proceeds/Rights
- Proceeds/Shares
- Proceeds/Warrants

For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

`adjustmentSplit` — Historical split pricing or volume adjustment
true | false

Historical split pricing or volume adjustment, specified as the comma-separated pair consisting of `'adjustmentSplit'` and a Boolean to reflect:

- Spin-Offs

- Stock Splits/Consolidations
- Stock Dividend/Bonus
- Rights Offerings/Entitlement

For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

`adjustmentFollowDPDF` — Historical pricing adjustment
true (default) | false

Historical pricing adjustment, specified as the comma-separated pair consisting of `'adjustmentFollowDPDF'` and a Boolean. Setting this name-value pair follows the **DPDF <GO>** option from the Bloomberg terminal. For details about these additional name-value pairs, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `logical`

Output Arguments

`d` — Bloomberg historical data
numeric array (default) | table | timetable

Bloomberg historical data, returned as a numeric array, table, or timetable. The data type of the historical data depends on the DataReturnFormat and DatetimeType properties of the connection object. The first column (or field) in the historical data contains the date. The remaining columns contain the requested data fields.

For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`sec` — Security list
cell array of character vectors

Security list, returned as a cell array of character vectors for the corresponding securities in `s`. The contents of `sec` are identical in value and order to `s`. You can return securities with any of the following identifiers:

- `buid`
- `cats`
- `cins`
- `common`
- `cusip`
- `isin`
- `sedol1`
- `sedol2`
- `sicovam`
- `svm`
- `ticker` (default)

- `wpk`

Tips

- You can check data and field availability by using the Bloomberg Excel Add-In.

Version History

Introduced in R2021a

See Also

`bloombergServer` | `close` | `getdata` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface” on page 5-39
“Retrieve Bloomberg Historical Data Using Bloomberg Server C++ Interface” on page 5-41

isconnection

Determine Bloomberg Server connection V3

Syntax

```
v = isconnection(c)
```

Description

`v = isconnection(c)` returns `true` (`1`) if `c` is a valid Bloomberg V3 connection using the Bloomberg Desktop C++ interface and `false` (`0`) otherwise.

Examples

Validate the Bloomberg Connection

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Validate the Bloomberg connection.

```
v = isconnection(c)
```

```
v =

     1
```

`v` returns `true` showing that the Bloomberg connection is valid.

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

Version History

Introduced in R2021a

See Also

`bloombergServer` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface” on page 5-39

lookup

Find information about securities for Bloomberg Server connection V3

Syntax

```
l = lookup(c,q,reqtype,Name,Value)
```

Description

`l = lookup(c,q,reqtype,Name,Value)` retrieves data based on criteria in the query `q` for a specific request type `reqtype` using the Bloomberg connection `c` with the Bloomberg Server C++ interface. For additional information about the query criteria and the possible name-value pair combinations, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Examples

Look Up Security

Create a Bloomberg connection, and then use the Security Lookup to retrieve information about the IBM corporate bond. For details about Bloomberg and the parameter values you can set, see the *Bloomberg API Developer's Guide* using the **WAPI<GO>** option from the Bloomberg terminal.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `lookup` function returns data as a structure.

```
c.DataReturnFormat = 'table';
```

Retrieve the instrument data for an IBM corporate bond with a maximum of 20 rows of data. The Security Lookup returns the security names and descriptions.

```
insts = lookup(c,'IBM','instrumentListRequest','maxResults',20, ...
    'yellowKeyFilter','YK_FILTER_CORP', ...
    'languageOverride','LANG_OVERRIDE_NONE');
```

Display the first three rows in the table. The first column contains the IBM corporate bond names, and the second column contains the bond descriptions.

```
insts(1:3,:)
```

```
ans =

  3×2 table

       security                    description
    ________________    ____________________________________

    'DD103619 <corp>'    'International Business Machines Corp'
    '459200AG <corp>'    'International Business Machines Corp'
    'EC767659 <corp>'    'International Business Machines Corp'
```

Close the Bloomberg connection.

```
close(c)
```

Look Up Curve

Use the Curve Lookup to retrieve information about the `'GOLD'` related curve `'CD1016'`. For details about Bloomberg and the parameter values you can set, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the curve data for the credit default swap subtype of corporate bonds for a `'GOLD'` related curve `'CD1016'`. Return a maximum of `10` rows of data for the U.S. with `'USD'` currency.

```
curves = lookup(c,'GOLD','curveListRequest','maxResults',10,...
                'countryCode','US','currencyCode','USD',...
                'curveid','CD1016','type','CORP','subtype','CDS')
```

```
curves =

            curve: {'YCCD1016 Index'}
      description: {'Goldman Sachs Group Inc/The'}
          country: {'US'}
         currency: {'USD'}
          curveid: {'CD1016'}
             type: {'CORP'}
          subtype: {'CDS'}
        publisher: {'Bloomberg'}
            bbgid: {''}
```

One row of data displays as Bloomberg curve name `'YCCD1016 Index'` with Bloomberg description `'Goldman Sachs Group Inc/The'` in the U.S. with `'USD'` currency. The Bloomberg short-form identifier for the curve is `'CD1016'`. Bloomberg is the publisher and the `bbgid` is blank.

Close the Bloomberg connection.

```
close(c)
```

Look Up Government Security

Use the Government Security Lookup to retrieve information for United States Treasury bonds. For details about Bloomberg and the parameter values you can set, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Filter government security data with ticker filter of `'T'` for a maximum of `10` rows of data.

```
govts = lookup(c,'T','govtListRequest','maxResults',10,...
               'partialMatch',false)
```

```
govts =

      parseky: {10x1 cell}
         name: {10x1 cell}
       ticker: {10x1 cell}
```

The Government Security Lookup returns `parseky` data, the name, and ticker of the United States Treasury bonds.

Display the `parseky` data.

```
govts.parseky
```

```
ans =
    '912828VS Govt'
    '912828RE Govt'
    '912810RC Govt'
    '912810RB Govt'
    '912828VU Govt'
    '912828VV Govt'
    '912828VB Govt'
    '912828VR Govt'
    '912828VW Govt'
    '912828VQ Govt'
```

Display the names of the United States Treasury bonds.

```
govts.name
```

```
ans =
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
    'United States Treasury Note/Bond'
```

Display the tickers of the United States Treasury bonds.

```
govts.ticker
```

```
ans =
    'T'
    'T'
    'T'
    'T'
    'T'
    'T'
    'T'
    'T'
    'T'
    'T'
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

q — Keyword query

character vector | string scalar | cell array of character vectors | string array

Keyword query, specified as a character vector, string scalar, cell array of character vectors, or string array. Each character vector or string denotes an item for which information is requested. For example, the keyword query can be a security, a curve type, or a filter ticker.

Data Types: `char` | `cell` | `string`

reqtype — Request type

`'instrumentListRequest'` | `'curveListRequest'` | `'govtListRequest'`

Request type, specified as the preceding values to denote the type of information request. `'instrumentListRequest'` denotes a security or instrument lookup request.

`'curveListRequest'` denotes a curve lookup request. `'govtListRequest'` denotes a government lookup request for government securities.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'maxResults'`, `20`, `'yellowKeyFilter'`, `'YK_FILTER_CORP'`, `'languageOverride'`, `'LANG_OVERRIDE_NONE'`, `'countryCode'`, `'US'`, `'currencyCode'`, `'USD'`, `'curveid'`, `'CD1016'`, `'type'`, `'CORP'`, `'subtype'`, `'CDS'`, `'partialMatch'`, `false`

`maxResults` — Number of rows in result data
numeric scalar

Number of rows in the result data, specified as the comma-separated pair consisting of `'maxResults'` and a numeric scalar to denote the total maximum number of rows of information to return. Result data can be one or more rows of data no greater than the number specified.

Data Types: `double`

`yellowKeyFilter` — Bloomberg yellow key filter
character vector | string scalar

Bloomberg yellow key filter, specified as the comma-separated pair consisting of `'yellowKeyFilter'` and a unique character vector or string scalar to denote the particular yellow key for government securities, corporate bonds, equities, and commodities, for example.

Data Types: `char` | `string`

`languageOverride` — Language override
character vector | string scalar

Language override, specified as the comma-separated pair consisting of `'languageOverride'` and a unique character vector or string scalar to denote a translation language for the result data.

Data Types: `char` | `string`

`countryCode` — Country code
character vector | string scalar

Country code, specified as the comma-separated pair consisting of `'countryCode'` and a character vector or string scalar to denote the country for the result data.

Data Types: `char` | `string`

`currencyCode` — Currency code
character vector | string scalar

Currency code, specified as the comma-separated pair consisting of `'currencyCode'` and a character vector or string scalar to denote the currency for the result data.

Data Types: `char` | `string`

`curveID` — Bloomberg short-form identifier for curve
character vector | string scalar

Bloomberg short-form identifier for a curve, specified as the comma-separated pair consisting of `'curveID'` and a character vector or string scalar.

Data Types: `char` | `string`

`type` — Bloomberg market sector type
character vector | string scalar

Bloomberg market sector type corresponding to the Bloomberg yellow keys, specified as the comma-separated pair consisting of `'type'` and a character vector or string scalar.

Data Types: `char` | `string`

`subtype` — Bloomberg market sector subtype
character vector | string scalar

Bloomberg market sector subtype, specified as the comma-separated pair consisting of `'subtype'` and a character vector or string scalar to further delineate the market sector type.

Data Types: `char` | `string`

`partialMatch` — Partial match on ticker
`true` | `false`

Partial match on ticker, specified as the comma-separated pair consisting of `'partialMatch'` and `true` or `false`. When set to `true`, you can filter securities by setting `q` to a query such as `'T*'`. When set to `false`, the securities are unfiltered.

Data Types: `logical`

Output Arguments

`l` — Lookup information
structure (default) | table

Lookup information, returned as a structure or table containing set properties depending on the request type. The data type of the lookup information depends on the DataReturnFormat property of the connection object.

For a list of the set properties and their descriptions, see the following tables.

'instrumentListRequest' Properties

Property	Description
`security`	Security name
`description`	Security long name

'curveListRequest' Properties

Property	Description
curve	Bloomberg curve name
description	Bloomberg description
country	Country code
currency	Currency code
curveid	Bloomberg short-form identifier for the curve
type	Bloomberg market sector type
subtype	Bloomberg market sector subtype
publisher	Bloomberg specified as publisher
bbgid	Bloomberg identifier

'govtListRequest' Properties

Property	Description
parsekey	Bloomberg security identifier (ticker or CUSIP, for example), price source, and source key (Bloomberg yellow key)
name	Government security name
ticker	Government security ticker

Version History

Introduced in R2021a

See Also

`bloombergServer` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface” on page 5-39

portfolio

Current portfolio data for Bloomberg Server connection V3

Syntax

```
d = portfolio(c,p,f)
d = portfolio(c,p,f,o,ov)
[d,plist] = portfolio( ___ )
```

Description

`d = portfolio(c,p,f)` returns current portfolio data for the fields `f` in the portfolio `p` using the `bloombergServer` object `c`.

`d = portfolio(c,p,f,o,ov)` returns current portfolio data using override field `o` and override value `ov`.

`[d,plist] = portfolio( ___ )` also returns the portfolio list `plist` using any of the input argument combinations in the previous syntaxes.

Examples

Request Portfolio Data

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Request portfolio data for a custom portfolio with portfolio identifier `U335877-1 Client`. Request data using all fields `f`.

```
p = 'U335877-1 Client';
f = {'PORTFOLIO_MEMBERS','PORTFOLIO_MPOSITION',...
     'PORTFOLIO_MWEIGHT','PORTFOLIO_DATA'};

d = portfolio(c,p,f)
```

```
d =

    PORTFOLIO_MPOSITION: {{0x1 cell}}
      PORTFOLIO_MWEIGHT: {{0x1 cell}}
```

```
        PORTFOLIO_DATA: {{0x1 cell}}
      PORTFOLIO_MEMBERS: {{0x1 cell}}
```

`d` is a structure that contains portfolio data. Each structure field corresponds to data for each portfolio field.

Close the connection.

```
close(c)
```

Request Portfolio Data Using Specific Date

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Request portfolio data for a custom portfolio with portfolio identifier `U335877-1 Client`. Request data using all fields `f`. Filter the portfolio data by specifying the date of November 3, 2014, using the override value `REFERENCE_DATE` equal to `20141103`.

```
p = 'U335877-1 Client';
f = {'PORTFOLIO_MEMBERS','PORTFOLIO_MPOSITION',...
     'PORTFOLIO_MWEIGHT','PORTFOLIO_DATA'};
o = {'REFERENCE_DATE'};
ov = {'20141103'};

[d,plist] = portfolio(c,p,f,o,ov)
```

```
d =

    PORTFOLIO_MPOSITION: {{0x1 cell}}
      PORTFOLIO_MWEIGHT: {{0x1 cell}}
         PORTFOLIO_DATA: {{0x1 cell}}
      PORTFOLIO_MEMBERS: {{0x1 cell}}

plist =

    'U335877-1 Client'
```

`d` is a structure that contains portfolio data. Each structure field corresponds to data for each portfolio field.

`plist` is a cell array that contains the portfolio identifier.

Close the connection.

```
close(c)
```

Input Arguments

c — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

p — Portfolio

character vector | string scalar

Portfolio, specified as a character vector or string scalar. Specify the portfolio by the ID that you can find in the upper-right corner of the portfolio display page. Append the text `' Client'` (without quotes) to the ID. For example, if the ID is `U335877-1`, then specify `'U335877-1 Client'`.

Access the portfolio display page by using the **PRTU<GO>** option from the Bloomberg terminal. For details, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `'U335877-1 Client'`

Data Types: `char` | `cell` | `string`

f — Portfolio fields

`'PORTFOLIO_DATA'` | `'PORTFOLIO_MEMBERS'` | `'PORTFOLIO_MPOSITION'` | `'PORTFOLIO_MWEIGHT'`

Portfolio fields, specified as one of the preceding values for one field. To specify multiple fields, use a cell array of these values.

Bloomberg Field Name	Bloomberg Field Description
`'PORTFOLIO_DATA'`	Returns a list of the identifiers, positions, market values, cost, cost date, and cost foreign exchange rate of each security in a custom portfolio.
`'PORTFOLIO_MEMBERS'`	Returns a list of identifiers for the members of a custom portfolio.
`'PORTFOLIO_MPOSITION'`	Returns a list of identifiers and the position for each security in a custom portfolio.
`'PORTFOLIO_MWEIGHT'`	Returns a list of identifiers and the percentage weight for each security in a custom portfolio.

Data Types: `char` | `cell`

o — Bloomberg override field

character vector | string scalar | cell array of character vectors | string array

Bloomberg override field, specified as a character vector, string scalar, cell array of character vectors, or string array. The Bloomberg value `'REFERENCE_DATE'` denotes returning Bloomberg data for a specific date.

Data Types: `char` | `cell` | `string`

ov — Bloomberg override field value

`[]` (default) | character vector | string scalar | cell array of character vectors | string array

Bloomberg override field value, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg override field value. A cell array of character vectors or string array denotes multiple Bloomberg override field values. Use this field value to filter the Bloomberg data result set.

Example: `'20100101'`

Data Types: `char` | `cell` | `string`

Output Arguments

d — Portfolio data

structure (default) | table

Portfolio data, returned as a structure or table. The data type of the portfolio data depends on the DataReturnFormat property of the connection object.

plist — Portfolio list

cell array of character vectors

Portfolio list, returned as a cell array of character vectors for the corresponding portfolio identifiers in `p`. The contents of `plist` are identical in value and order to `p`.

Version History

Introduced in R2021a

See Also

`bloombergServer` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

“Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface” on page 5-39

realtime

Real-time data for Bloomberg Server connection V3

Syntax

```
d = realtime(c,s,f)
[~,t] = realtime(c,s,f,eventhandler)
```

Description

`d = realtime(c,s,f)` returns the data for the `bloombergServer` object `c` with the Bloomberg Server C++ interface, security list `s`, and requested fields `f`. `realtime` accesses the Bloomberg Market Data service.

`[~,t] = realtime(c,s,f,eventhandler)` returns an empty output and the timer `t` associated with the real-time event handler for the subscription list. Given connection `c`, the `realtime` function subscribes to a security or securities `s` and requests fields `f`, to update in real time while running an event handler `eventhandler`.

Examples

Retrieve Data for One Security

Retrieve a snapshot of data for one security only.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the last trade and volume of the IBM security.

```
d = realtime(c,'IBM US Equity',{'Last_Trade','Volume'})
```

```
d =

    LAST_TRADE: '181.76'
        VOLUME: '7277793'
```

Close the Bloomberg connection.

```
close(c)
```

Retrieve Data for One Security Using Event Handler

You can create your own event handler function to process Bloomberg data. For this example, use the event handler `disp` that displays Bloomberg stock tick data at the command line.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the last price and volume for the IBM security using the event handler `disp`.

```
[~,t] = realtime(c,'IBM US Equity',{'LAST_PRICE','VOLUME'}, ...
    'disp')
```

```
t =


   Timer Object: timer-4

   Timer Settings
      ExecutionMode: fixedRate
              Period: 0.05
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: 1x5 cell array
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''

   Columns 1 through 4

    {'SecurityID'   }    {'LAST_PRICE'}    {'SecurityID'   }    {'VOLUME'}
    {'IBM US Equity'}    {'118.490000'}    {'IBM US Equity'}    {'744066'}
...
```

`realtime` returns the MATLAB timer object with its properties. Then, `realtime` returns the stock tick data for the IBM security with the last price and volume.

Stop the display of real-time data.

```
stop(t)
c.Session.stopSubscriptions
```

Close the Bloomberg connection.

```
close(c)
```

Retrieve Data for Multiple Securities Using Event Handler

You can create your own event handler function to process Bloomberg data. For this example, use the event handler `disp` that returns Bloomberg stock tick data at the command line.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the last price and volume for IBM and Ford Motor Company securities.

```
[~,t] = realtime(c,{'IBM US Equity','F US Equity'}, ...
    {'LAST_PRICE','VOLUME'},'disp')

t =


   Timer Object: timer-4

   Timer Settings
      ExecutionMode: fixedRate
              Period: 0.05
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: 1x5 cell array
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''

Columns 1 through 6

    {'SecurityID' }    {'LAST_PRICE'}    {'SecurityID' }    {'VOLUME'  }    {'SecurityID'   }
    {'F US Equity'}    {'8.960000'  }    {'F US Equity'}    {'13423731'}    {'IBM US Equity'}

  Columns 7 through 8

    {'SecurityID'   }    {'VOLUME'}
    {'IBM US Equity'}    {'744066'}
...
```

`realtime` returns the MATLAB timer object with its properties. Then, `realtime` returns the stock tick data for the IBM and Ford Motor Company securities with the last price and volume.

Stop the display of real-time data.

```
stop(t)
c.Session.stopSubscriptions
```

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

s — Security list

character vector | string scalar | cell array of character vectors | string array

Security list, specified as a character vector or string scalar for one security or a cell array of character vectors or string array for multiple securities. You can specify the security by name or by CUSIP, and with or without the pricing source.

Data Types: `char` | `cell` | `string`

f — Bloomberg data fields

character vector | string scalar | cell array of character vectors | string array

Bloomberg data fields, specified as a character vector, string scalar, cell array of character vectors, or string array. A character vector or string denotes one Bloomberg data field name. A cell array of character vectors or string array denotes multiple Bloomberg data field names. For details about the fields you can specify, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Example: `{'LAST_PRICE';'OPEN'}`

Data Types: `char` | `cell` | `string`

eventhandler — Event handler

character vector | string scalar

Event handler, specified as a character vector or string scalar that denotes the name of an event handler function that you define. You can define an event handler function to process any type of real-time Bloomberg events. The specified event handler function runs every time the timer fires.

Data Types: `char` | `string`

Output Arguments

d — Bloomberg data

structure (default) | table | timetable

Bloomberg data, returned as a structure, table, or timetable. The data type of the Bloomberg data depends on the DataReturnFormat and DatetimeType properties of the connection object. For details

about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

`t` — MATLAB timer
object

MATLAB timer, returned as a MATLAB object. For details about this object, see `timer`.

Version History

Introduced in R2021a

See Also

`bloombergServer` | `close` | `getdata` | `history` | `timeseries`

Topics

"Retrieve Bloomberg Real-Time Data Using Bloomberg Server C++ Interface" on page 5-47
"Writing and Running Custom Event Handler Functions" on page 1-26

tahistory

Historical technical analysis for Bloomberg Server connection V3

Syntax

```
d = tahistory(c)
d = tahistory(c,s,startdate,enddate,study,period,Name,Value)
```

Description

`d = tahistory(c)` returns the Bloomberg Server V3 session technical analysis data study and element definitions.

`d = tahistory(c,s,startdate,enddate,study,period,Name,Value)` returns the Bloomberg V3 session technical analysis data study and element definitions with additional options specified by one or more name-value pair arguments.

Examples

Request Bloomberg Directional Movement Indicator (DMI) Study for Security

Return all available Bloomberg studies and use the DMI study to run a technical analysis for a security.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

List the available Bloomberg studies.

```
d = tahistory(c)
```

```
d =

          dmiStudyAttributes: [1x1 struct]
        smavgStudyAttributes: [1x1 struct]
         bollStudyAttributes: [1x1 struct]
          maoStudyAttributes: [1x1 struct]
           fgStudyAttributes: [1x1 struct]
          rsiStudyAttributes: [1x1 struct]
         macdStudyAttributes: [1x1 struct]
```

```
          tasStudyAttributes: [1x1 struct]
        emavgStudyAttributes: [1x1 struct]
       maxminStudyAttributes: [1x1 struct]
         ptpsStudyAttributes: [1x1 struct]
         cmciStudyAttributes: [1x1 struct]
         wlprStudyAttributes: [1x1 struct]
        wmavgStudyAttributes: [1x1 struct]
      trenderStudyAttributes: [1x1 struct]
          gocStudyAttributes: [1x1 struct]
         kltnStudyAttributes: [1x1 struct]
    momentumStudyAttributes: [1x1 struct]
         rocStudyAttributes: [1x1 struct]
         maeStudyAttributes: [1x1 struct]
        hurstStudyAttributes: [1x1 struct]
         chkoStudyAttributes: [1x1 struct]
           teStudyAttributes: [1x1 struct]
        vmavgStudyAttributes: [1x1 struct]
        tmavgStudyAttributes: [1x1 struct]
          atrStudyAttributes: [1x1 struct]
          rexStudyAttributes: [1x1 struct]
          adoStudyAttributes: [1x1 struct]
           alStudyAttributes: [1x1 struct]
          etdStudyAttributes: [1x1 struct]
          vatStudyAttributes: [1x1 struct]
         tvatStudyAttributes: [1x1 struct]
           pdStudyAttributes: [1x1 struct]
           rvStudyAttributes: [1x1 struct]
      ipmavgStudyAttributes: [1x1 struct]
       pivotStudyAttributes: [1x1 struct]
          orStudyAttributes: [1x1 struct]
         pcrStudyAttributes: [1x1 struct]
          bsStudyAttributes: [1x1 struct]
```

`d` contains structures pertaining to each available Bloomberg study.

Display the name-value pairs for the DMI study.

```
d.dmiStudyAttributes
```

```
ans =

                period: [1x104 char]
     priceSourceHigh: [1x123 char]
      priceSourceLow: [1x121 char]
    priceSourceClose: [1x125 char]
```

Obtain more information about the `period` property.

```
d.dmiStudyAttributes.period
```

```
ans =

DEFINITION period {

    Min Value = 1

    Max Value = 1

    TYPE Int64
```

```
} // End Definition: period
```

Run the DMI study for the IBM security for the last month with `period` equal to `14`, the high price, the low price, and the closing price.

```
d = tahistory(c,'IBM US Equity',floor(now)-30,floor(now),'dmi',...
              'all_calendar_days','period',14,...
              'priceSourceHigh','PX_HIGH',...
              'priceSourceLow','PX_LOW','priceSourceClose','PX_LAST')
```

```
d =

         date: [31x1 double]
     DMI_PLUS: [31x1 double]
    DMI_MINUS: [31x1 double]
          ADX: [31x1 double]
         ADXR: [31x1 double]
```

`d` contains a `studyDataTable` with one `studyDataRow` for each interval returned.

Display the first five dates in the returned data.

```
d.date(1:5,1)
```

```
ans =

     735507.00
     735508.00
     735509.00
     735510.00
     735511.00
```

Display the first five prices in the plus DI line.

```
d.DMI_PLUS(1:5,1)
```

```
ans =

         18.92
         17.84
         16.83
         15.86
         15.63
```

Display the first five prices in the minus DI line.

```
d.DMI_MINUS(1:5,1)
```

```
ans =

         30.88
         29.12
         28.16
         30.67
         29.24
```

Display the first five values of the Average Directional Index.

```
d.ADX(1:5,1)
```

```
ans =

         22.15
         22.28
         22.49
         23.15
         23.67
```

Display the first five values of the Average Directional Movement Index Rating.

```
d.ADXR(1:5,1)
```

```
ans =

         25.20
         25.06
         25.05
         25.60
         26.30
```

Close the Bloomberg connection.

```
close(c)
```

Request DMI Study for Security with Pricing Source

Run a technical analysis to return the DMI study for a security with a pricing source.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Run the DMI study for the Microsoft security with pricing source `ETPX` for the last month with `period` equal to `14`, the high price, the low price, and the closing price.

```
d = tahistory(c,'MSFT@ETPX US Equity',floor(now)-30,floor(now),...
              'dmi','all_calendar_days','period',14,...
              'priceSourceHigh','PX_HIGH','priceSourceLow','PX_LOW',...
              'priceSourceClose','PX_LAST')
```

```
d =

          date: [31x1 double]
     DMI_PLUS: [31x1 double]
    DMI_MINUS: [31x1 double]
          ADX: [31x1 double]
         ADXR: [31x1 double]
```

`d` contains a `studyDataTable` with one `studyDataRow` for each interval returned.

Display the first five dates in the returned data.

```
d.date(1:5,1)
```

```
ans =

     735507.00
     735508.00
     735509.00
     735510.00
     735511.00
```

Display the first five prices in the plus DI line.

```
d.DMI_PLUS(1:5,1)
```

```
ans =

         28.37
         30.63
         32.72
         30.65
         29.37
```

Display the first five prices in the minus DI line.

```
d.DMI_MINUS(1:5,1)
```

```
ans =

         21.97
         21.17
         19.47
         18.24
         17.48
```

Display the first values of the Average Directional Index.

```
d.ADX(1:5,1)
```

```
ans =

         13.53
         13.86
         14.69
         15.45
         16.16
```

Display the first five values of the Average Directional Movement Index Rating.

```
d.ADXR(1:5,1)
```

```
ans =

         15.45
         15.36
         15.53
```

```
    15.85
    16.37
```

Close the Bloomberg connection.

```
close(c)
```

Return DMI Study Data as Table with Dates

Create a Bloomberg connection, and then return data for a DMI study. The `tahistory` function returns data for dates as a `datetime` array.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `tahistory` function returns data as a structure.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Run the DMI study for the IBM security from June 12, 2017, through June 16, 2017, with `period` equal to 14, the high price, the low price, and the closing price.

```
d = tahistory(c,'IBM US Equity','6/12/2017','6/16/2017','dmi', ...
    'all_calendar_days','period',14,'priceSourceHigh','PX_HIGH', ...
    'priceSourceLow','PX_LOW','priceSourceClose','PX_LAST');
```

Access the DMI study data for the first three dates.

```
d(1:3,:)


ans =

  3×5 table

       date        DMI_PLUS    DMI_MINUS     ADX      ADXR
    ___________    ________    _________    _____    _____
```

```
    12-Jun-2017     30.48        16.31      33.93    45.26
    13-Jun-2017     28.88        15.45      33.67    44.10
    14-Jun-2017     26.62        18.98      32.46    42.67
```

`d` is a `table` that contains these columns:

- `date` -- Date
- `DMI_PLUS` -- Prices in plus DI line
- `DMI_MINUS` -- Prices in minus DI line
- `ADX` -- Average Directional Index values
- `ADXR` -- Average Directional Movement Index Rating values

Access the first three dates in the returned data.

```
d.date(1:3)
```

```
ans =

  3×1 datetime array

   12-Jun-2017
   13-Jun-2017
   14-Jun-2017
```

Close the Bloomberg connection.

```
close(c)
```

Return DMI Study Data as Timetable

Create a Bloomberg connection, and then return data for a DMI study. The `tahistory` function returns data as a `timetable`.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `tahistory` function returns data as a structure.

```
c.DataReturnFormat = 'timetable';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Run the DMI study for the IBM security from June 12, 2017 through June 16, 2017 with `period` equal to 14, the high price, the low price, and the closing price.

```
d = tahistory(c,'IBM US Equity','6/12/2017','6/16/2017','dmi', ...
    'all_calendar_days','period',14,'priceSourceHigh','PX_HIGH', ...
    'priceSourceLow','PX_LOW','priceSourceClose','PX_LAST');
```

Access the DMI study data for the first three dates.

```
d(1:3,:)
```

```
ans =

  3×4 timetable

       date         DMI_PLUS    DMI_MINUS     ADX     ADXR
    ____________    ________    _________    _____    _____

    12-Jun-2017      30.48         16.31      33.93    45.26
    13-Jun-2017      28.88         15.45      33.67    44.10
    14-Jun-2017      26.62         18.98      32.46    42.67
```

`d` is a `timetable` that contains these columns:

- `date` -- Date
- `DMI_PLUS` -- Prices in plus DI line
- `DMI_MINUS` -- Prices in minus DI line
- `ADX` -- Average Directional Index values
- `ADXR` -- Average Directional Movement Index Rating values

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

`s` — Security

character vector | string scalar

Security, specified as a character vector or string scalar for a single Bloomberg security.

Data Types: `char` | `string`

`startdate` — Start date

numeric scalar | character vector | string scalar

Start date, specified as a numeric scalar, character vector, or string scalar to denote the start date of the date range for the returned tick data.

Example: `floor(now-1)`

Data Types: `double` | `char` | `string`

`enddate` — End date
numeric scalar | character vector | string scalar

End date, specified as a numeric scalar, character vector, or string scalar to denote the end date of the date range for the returned tick data.

Example: `floor(now)`

Data Types: `double` | `char` | `string`

`study` — Study type
character vector | string scalar

Study type, specified as a character vector or string scalar to denote the study to use for historical analysis.

Data Types: `char` | `string`

`period` — Periodicity
`'daily'` | `'weekly'` | `'monthly'` | `'quarterly'` | ...

Periodicity, specified as one of these values to denote the data to return. For specifying multiple values, use a cell array. For example, when `period` is set to `{'daily','all_calendar_days'}`, `tahistory` returns daily data for all calendar days, and reports missing data as `NaN`s. When `period` is set to `'active_days_only'`, `tahistory` returns data using the default periodicity for active trading days only. The default periodicity depends on the security. If a security is reported on a monthly basis, the default periodicity is monthly. These tables show the values for `period`.

To specify the periodicity of the return data, see this table.

Value	Description
`'daily'`	Return data for each day.
`'weekly'`	Return data for each week.
`'monthly'`	Return data for each month.
`'quarterly'`	Return data for each quarter.
`'semi_annually'`	Return data semiannually.
`'yearly'`	Return data for each year.

The anchor date is the date to which all other reported dates are related. To specify the anchor date, see this table.

Value	Description
`'actual'`	Anchor date specification for an actual date. For this function, for periodicities other than daily, `enddate` is the anchor date. If the period is weekly and the `enddate` is a Thursday, every data point is a Thursday, or the nearest prior business day to Thursday. If the period is monthly and the `enddate` is the 20th of a month, every data point is the 20th of each month in the date range.
`'calendar'`	Anchor date specification for a calendar year.
`'fiscal'`	Anchor date specification for a fiscal year.

To specify returning data for particular days, see this table.

Value	Description
`'non_trading_weekdays'`	Return data for all weekdays.
`'all_calendar_days'`	Return data for all calendar days.
`'active_days_only'`	Return data for only active trading days.

To specify how to fill missing values, see this table.

Value	Description
`'previous_value'`	Fill missing values with previous values for dates without trading activity for the security.
`'nil_value'`	Fill missing values with a `NaN` for dates without trading activity for the security.

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'period',14,'priceSourceHigh','PX_HIGH','priceSourceLow','PX_LOW','priceSourceClose','PX_LAST'`

Note For details about the full list of name-value pair arguments, see the Bloomberg tool located at `C:\blp\API\APIv3\bin\BBAPIDemo.exe`.

`period` — Period

numeric scalar

Period, specified as the comma-separated pair consisting of `'period'` and a numeric scalar. For details about the period, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `double`

`priceSourceHigh` — High price
character vector | string scalar

High price, specified as the comma-separated pair consisting of `'priceSourceHigh'` and a character vector or string scalar. For details about the high price, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

`priceSourceLow` — Low price
character vector | string scalar

Low price, specified as the comma-separated pair consisting of `'priceSourceLow'` and a character vector or string scalar. For details about the low price, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

`priceSourceClose` — Closing price
character vector | string scalar

Closing price, specified as the comma-separated pair consisting of `'priceSourceClose'` and a character vector or string scalar. For details about the closing price, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Data Types: `char` | `string`

Output Arguments

`d` — Technical analysis data
structure (default) | table | timetable

Technical analysis data, returned as a structure, table, or timetable. The data type of the returned data depends on the DataReturnFormat and DatetimeType properties of the connection object.

For details about the data, see the *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Version History

Introduced in R2021a

See Also

`bloombergServer` | `close` | `getdata` | `history` | `realtime` | `timeseries`

Topics

"Retrieve Bloomberg Current Data Using Bloomberg Server C++ Interface" on page 5-39

timeseries

Intraday tick data for Bloomberg Server connection V3

Syntax

```
d = timeseries(c,s,date)
d = timeseries(c,s,date,interval,field)
d = timeseries(c,s,date,[],field,options,values)

d = timeseries(c,s,{startdate,enddate})
d = timeseries(c,s,{startdate,enddate},interval,field)
d = timeseries(c,s,{startdate,enddate},[],field)
d = timeseries(c,s,{startdate,enddate},[],field,options,values)

d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval)
d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval,field)

d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime},
interval)
d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime},
interval,field)
```

Description

`d = timeseries(c,s,date)` retrieves raw tick data using the `bloombergServer` object with the Bloomberg Server C++ interface and a security for a specific date.

`d = timeseries(c,s,date,interval,field)` retrieves raw tick data that is aggregated into intervals for a specific field.

`d = timeseries(c,s,date,[],field,options,values)` retrieves raw tick data without an aggregation interval for a specific field with the specified options and corresponding values.

`d = timeseries(c,s,{startdate,enddate})` retrieves raw tick data for a date range using a start date and an end date.

`d = timeseries(c,s,{startdate,enddate},interval,field)` retrieves raw tick data for a specific date range aggregated into intervals for a specific field.

`d = timeseries(c,s,{startdate,enddate},[],field)` retrieves raw tick data for a specific date range without an aggregation interval for a specific field.

`d = timeseries(c,s,{startdate,enddate},[],field,options,values)` retrieves raw tick data for a specific date range without an aggregation interval for a specific field with specified options and corresponding values.

`d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval)` retrieves raw trade tick data for a specific time range for each day within a specific date range, aggregated into intervals.

`d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval,field)` uses a specific field for tick data to return.

`d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime}, interval)` retrieves raw trade tick data for a whole day increment within a specific date and time range, aggregated into intervals.

`d = timeseries(c,s,{startdate:dayincrement:enddate,starttime,endtime}, interval,field)` uses a specific field for tick data to return.

Examples

Retrieve Tick Data for Specific Date and Pricing Source

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date. Use a security with and without a pricing source to retrieve tick data.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the trade tick series using the IBM security for today.

```
d = timeseries(c,'IBM US Equity',floor(now))
```

```
d =

    'TRADE'    [735537.40]    [181.69]    [100.00]
    'TRADE'    [735537.40]    [181.69]    [100.00]
    'TRADE'    [735537.40]    [181.68]    [100.00]
    ...
```

The columns in `d` are:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size

Here, the first row shows that 100 IBM shares sold for $181.69 today.

Retrieve the trade tick series using the Microsoft security with pricing source `ETPX` for today.

```
d = timeseries(c,'MSFT@ETPX US Equity',floor(now))
```

```
d =

    'TRADE'    [735537.40]    [35.53]    [100.00]
    'TRADE'    [735537.40]    [35.55]    [200.00]
    'TRADE'    [735537.40]    [35.55]    [100.00]
    ...
```

Here, the first row shows that 100 Microsoft shares are sold for $35.53 today.

Close the Bloomberg connection.

```
close(c)
```

Time Interval with Specific Field

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date. Specify the tick data to return using a time interval and field.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the trade tick series using the IBM security aggregated into 5-minute intervals for today.

```
d = timeseries(c,'IBM US Equity',floor(now),5,'Trade')
```

```
d =

  Columns 1 through 7

   735537.40        181.69        181.99        180.10        181.84     252322.00        861.00
   735537.40        181.90        181.97        181.57        181.65      78570.00        535.00
   735537.40        181.73        182.18        181.58        182.07     124898.00        817.00
     ...

  Column 8

   45815588.00
   14282076.00
   22710954.00
   ...
```

The columns in `d` contain the following:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price

- Volume of ticks
- Number of ticks
- Total tick value in the bar

Here, the first row of data shows prices and tick data for the current date. The next row shows tick data for 5 minutes later.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Tick Data Using Option and Value

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date and field. Use option and value to return additional data.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the trade tick series using the `'F US Equity'` security without specifying the aggregation parameter for today. Also, return the condition codes.

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               'includeConditionCodes','true')
```

```
d =

    'TRADE'    [735556.57]    [17.12]    [   100.00]    'R6,IS'
    'TRADE'    [735556.57]    [17.12]    [   100.00]    ''
    'TRADE'    [735556.57]    [17.12]    [   500.00]    ''
    ...
```

The columns in `d` contain the following:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size
- Condition codes

Here, the first row shows that 100 `'F US Equity'` security shares sold for $17.12 today.

Close the Bloomberg connection.

```
close(c)
```

Retrieve Tick Data Using Date Range

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the tick series for the `'F US Equity'` security for the last business day from the beginning of the day to noon.

```
d = timeseries(c,'F US Equity',{floor(now-4),floor(now-3.5)})
```

```
d =

    'TRADE'    [735552.67]    [17.09]    [   200.00]
    'TRADE'    [735552.67]    [17.09]    [   100.00]
    'TRADE'    [735552.67]    [17.09]    [   100.00]
   ...
```

The columns in `d` are:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size

Here, the first row shows that 200 `'F US Equity'` security shares were sold for $17.09 on the last business day.

Close the Bloomberg connection.

```
close(c)
```

Date Range with Interval and Specific Field

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range. Specify the interval and field.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the trade tick series for the past 50 days for the IBM security aggregated into 5-minute intervals.

```
d = timeseries(c,'IBM US Equity',{floor(now)-50,floor(now)},5,'Trade')
```

```
ans =

  Columns 1 through 7

   735487.40        187.20        187.60        187.02        187.08     207683.00        560.00
   735487.40        187.03        187.13        186.65        186.78      46990.00        349.00
   735487.40        186.78        186.78        186.40        186.47      51589.00        399.00
     ...

Column 8

   38902968.00
    8779374.00
    9626896.00
   ...
```

The columns in `d` contain the following:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row of data shows prices and tick data for the current date. The next row shows tick data for 5 minutes later.

Close the Bloomberg connection.

```
close(c)
```

Date Range with Numerous Fields

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range and numerous fields.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return the Bid, Ask, and trade tick series for the security `'F US Equity'` for yesterday with a time interval at noon, without specifying the aggregation parameter.

```
d = timeseries(c,'F US Equity',{floor(now-1)+.5,floor(now-1)+.51},...
               [],{'Bid','Ask','Trade'})

d =

    'TRADE'    [735550.50]    [16.71]    [100.00]
    'ASK'      [735550.50]    [16.71]    [312.00]
    'BID'      [735550.50]    [16.70]    [177.00]
    ...
```

The columns in `d` are:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size

Here, the first row shows that 100 `'F US Equity'` security shares sold for $16.71 yesterday.

Close the Bloomberg connection.

```
close(c)
```

Date Range with Options and Values

First, create a Bloomberg Desktop connection. Then, retrieve tick data for a specific date range. Specify options and values to return additional data.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return the trade tick series for the security `'F US Equity'` for yesterday with a time interval at noon, without specifying the aggregation parameter. Also, return the condition codes, exchange codes, and broker codes.

```
d = timeseries(c,'F US Equity',{floor(now-1)+.5,floor(now-1)+.51},...
               [],'Trade',{'includeConditionCodes',...
               'includeExchangeCodes','includeBrokerCodes'},...
               {'true','true','true'})
```

```
d =

    'TRADE'    [735550.50]    [16.71]    [100.00]    'T'     'D'
    'TRADE'    [735550.50]    [16.70]    [400.00]    'IS'    'B'
    'TRADE'    [735550.50]    [16.70]    [100.00]    'IS'    'B'
    ...
```

The columns in `d` contain the following:

- Tick type
- Numeric representation of the date and time
- Tick value
- Tick size
- Exchange condition codes
- Exchange codes

Broker codes are available for Canadian, Finnish, Mexican, Philippine, and Swedish equities only. In this case, the broker buy code appears in the seventh column and the broker sell code appears in the eighth column.

Here, the first row shows that 100 `'F US Equity'` security shares sold for $16.71 yesterday.

Close the Bloomberg connection.

```
close(c)
```

Date and Time Range with Interval

Use Bloomberg to retrieve raw trade tick data by specifying a time range for each day in a specific date range. Specify the time interval for the tick data.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the trade tick series for the `'F US Equity'` security for the last two days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. `d` is a numeric matrix.

```
s = 'F US Equity';
startdate = datetime('today')-1;
enddate = datetime('today');
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;

d = timeseries(c,s,{startdate:enddate,starttime,endtime},interval);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    736959.40         11.71         11.81         11.71         11.79
    736959.40         11.79         11.81         11.75         11.79
    736959.40         11.80         11.82         11.78         11.80

  Columns 6 through 8

   1375547.00       1190.00   16196757.00
    598924.00        898.00    7058724.00
    488655.00        641.00    5768371.50
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

Determine the maximum high price for the last two days.

```
highprices = d(:,3);
m = max(highprices)
```

```
m =

         11.82
```

Close the Bloomberg connection.

```
close(c)
```

Date and Time Range with Interval and Specific Field

Use Bloomberg to retrieve raw tick data by specifying a time range for each day in a specific date range. Specify the time interval and the field for the type of tick data to return. Here, specify the bid tick data.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the tick series for the `'F US Equity'` security for the last two days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. Specify retrieving the bid tick series. `d` is a numeric matrix.

```
s = 'F US Equity';
startdate = datetime('today')-1;
enddate = datetime('today');
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;
field = 'BID';

d = timeseries(c,s, ...
    {startdate:enddate,starttime,endtime}, ...
    interval,field);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)


ans =
```

```
  Columns 1 through 5

     736959.40         11.70         11.80         11.70         11.79
     736959.40         11.79         11.80         11.75         11.79
     736959.40         11.79         11.81         11.78         11.80

  Columns 6 through 8

     397711.00       1442.00    4681704.50
     450997.00       1698.00    5311330.50
     464761.00       1391.00    5481707.50
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

Determine the maximum high price for the last two days.

```
highprices = d(:,3);
m = max(highprices)
```

```
m =

          11.81
```

Close the Bloomberg connection.

```
close(c)
```

Date and Time Range with Day Increment and Interval

Use Bloomberg to retrieve raw trade tick data by specifying a time range for each day in a specific date range. Specify a day increment for the date range and the time interval for the tick data.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the trade tick series for the `'IBM US Equity'` security for the last two months. Set the day increment to 5 days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. `d` is a numeric matrix.

```
s = 'IBM US Equity';
startdate = datetime('today')-60;
enddate = datetime('today');
dayincrement = 5;
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;

d = timeseries(c,s, ...
    {startdate:dayincrement:enddate,starttime,endtime}, ...
    interval);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)
```

```
ans =

  Columns 1 through 5

    736900.40        147.00        147.04        146.55        146.62
    736900.40        146.62        146.87        146.62        146.71
    736900.40        146.72        146.79        146.52        146.54

  Columns 6 through 8

    125558.00        393.00   18440146.00
     39535.00        258.00    5800969.00
     49659.00        314.00    7282961.00
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks

- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

After the tick data for the first day in the date range, `d` contains tick data for a trading day that is 5 days later.

Close the Bloomberg connection.

```
close(c)
```

Date and Time Range with Day Increment, Interval, and Specific Field

Use Bloomberg to retrieve raw tick data by specifying a time range for each day in a specific date range. Specify a day increment for the date range, the time interval for the tick data, and the field for the type of tick data to return. Here, specify the bid tick data.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Retrieve the trade tick series for the `'F US Equity'` security for the last two months. Set the day increment to 5 days. Use the time range from the beginning of the trading day through noon. Retrieve tick data aggregated into 5-minute intervals. Specify the bid tick series. `d` is a numeric matrix.

```
s = 'F US Equity';
startdate = datetime('today')-60;
enddate = datetime('today');
dayincrement = 5;
starttime = "09:30:00";
endtime = "12:00:00";
interval = 5;
field = 'BID';

d = timeseries(c,s, ...
    {startdate:dayincrement:enddate,starttime,endtime}, ...
    interval,field);
```

Set the display output for currency.

```
format bank
```

Display the first three ticks.

```
d(1:3,:)


ans =

  Columns 1 through 5

     736900.40         11.50         11.54         11.49         11.50
     736900.40         11.50         11.50         11.48         11.48
     736900.40         11.48         11.49         11.44         11.44

  Columns 6 through 8

     422305.00       1158.00    4863894.00
     575966.00       1180.00    6617854.00
     288147.00       1489.00    3305491.75
```

The columns in `d` are:

- Numeric representation of date and time
- Open price
- High price
- Low price
- Closing price
- Volume of ticks
- Number of ticks
- Total tick value in the bar

The first row shows tick data at the start time of the time range. The next row shows tick data for 5 minutes later.

After the tick data for the first day in the date range, `d` contains tick data for a trading day that is 5 days later.

Close the Bloomberg connection.

```
close(c)
```

Return Tick Data as Table with Dates

Create a Bloomberg connection, and then return intraday tick data. The `timeseries` function returns data for dates as a `datetime` array.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return data as a table by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `timeseries` function returns data as a numeric array.

Return dates as a `datetime` array by setting the `DatetimeType` property of the connection object. In this case, the table contains dates in variables that are `datetime` arrays.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve the trade tick series for the IBM® security aggregated into 5-minute intervals for today. `d` is a table that contains the tick series data.

```
s = 'IBM US Equity';
date = floor(now);
interval = 5;
field = 'Trade';

d = timeseries(c,s,date,interval,field);
```

Access the first three ticks of data.

```
d(1:3,:)
```

```
ans =

  3×8 table

      DATE         OPEN      HIGH      LOW      CLOSE      VOLUME      NUMBER_OF_TICKS    TOTAL_
    ___________    ______    ______    ______    ______    _________    _______________    ______

    21-Dec-2017    153.17    153.31    153.08    153.31    152524.00        442.00         23367
    21-Dec-2017    153.35    153.35    152.82    152.84     46051.00        291.00          7048
    21-Dec-2017    152.84    153.21    152.82    153.16     30966.00        225.00          4737
```

`d` contains columns with the following data:

- Date
- Open price
- High price
- Low price
- Closing price
- Volume
- Number of ticks

- Total tick value in the bar

Access the first three dates in the `DATE` column.

```
d.DATE(1:3)
```

```
ans =

  3×1 datetime array

   21-Dec-2017
   21-Dec-2017
   21-Dec-2017
```

Close the Bloomberg connection.

```
close(c)
```

Return Tick Data as Timetable

Create a Bloomberg connection, and then return intraday tick data. The `timeseries` function returns data for dates as a `timetable`.

Connect to the Bloomberg Server using the IP address of the machine running the Bloomberg Server. This example uses the Bloomberg Server C++ interface and assumes the following:

- The Bloomberg UUID is `12345678`.
- The IP address for the machine running the Bloomberg Server is `'111.11.11.111'`.

`c` is a `bloombergServer` object.

```
uuid = 12345678;
ipaddress = '111.11.11.111';

c = bloombergServer(uuid,ipaddress);
```

Return data as a `timetable` by setting the `DataReturnFormat` property of the connection object. If you do not set this property, the `timeseries` function returns data as a numeric array.

```
c.DataReturnFormat = 'timetable';
```

Adjust the display format of the returned data for currency.

```
format bank
```

Retrieve the trade tick series for the IBM® security aggregated into 5-minute intervals for today. `d` is a `timetable` that contains the tick series data.

```
s = 'IBM US Equity';
date = floor(now);
interval = 5;
field = 'Trade';

d = timeseries(c,s,date,interval,field);
```

Access the first three ticks of data.

```
d(1:3,:)
```

```
ans =

  3×7 timetable

      DATE         OPEN      HIGH      LOW      CLOSE      VOLUME      NUMBER_OF_TICKS    TOTAL_
    ___________    ______    ______    ______    ______    _________    _______________    ______

    21-Dec-2017    153.17    153.31    153.08    153.31    152524.00        442.00         23367
    21-Dec-2017    153.35    153.35    152.82    152.84     46051.00        291.00          7048
    21-Dec-2017    152.84    153.21    152.82    153.16     30966.00        225.00          4737
```

`d` is a `timetable` that contains the following data:

- Date
- Open price
- High price
- Low price
- Closing price
- Volume
- Number of ticks
- Total tick value in the bar

Close the Bloomberg connection.

```
close(c)
```

Input Arguments

c — Bloomberg Server connection

`bloombergServer` object

Bloomberg Server connection, specified as a `bloombergServer` object.

s — Security

character vector | string scalar

Security, specified as a character vector or string scalar for a single Bloomberg security.

Data Types: `char` | `string`

date — Date

numeric scalar | character vector | string scalar | `datetime` array

Date, specified as a numeric scalar, character vector, string scalar, or `datetime` array. `date` specifies the date for the returned tick data based on the entire day from midnight until 11:59:59 p.m.

Example: `floor(now)`

Data Types: `double` | `char` | `string` | `datetime`

`interval` — Time interval

numeric scalar

Time interval, specified as a numeric scalar to denote the number of minutes between ticks for the returned tick data.

Data Types: `double`

`field` — Bloomberg field

`'TRADE'` (default) | `'BID'` | `'ASK'` | ...

Bloomberg field, specified as one of these values that define the tick data to return.

Request Type	Valid Bloomberg Field Values
IntradayBarRequest with time interval specified	`'TRADE'`
	`'BID'`
	`'ASK'`
	`'BID_BEST'`
	`'ASK_BEST'`
IntradayTickRequest with no time interval specified	`'TRADE'`
	`'BID'`
	`'ASK'`
	`'BID_BEST'`
	`'ASK_BEST'`
	`'SETTLE'`

`options` — Bloomberg API options

`'includeConditionCodes'` | `'includeExchangeCodes'` | `'includeBrokerCodes'` | ...

Bloomberg API options, specified as one of the values in this table.

Value	Description
`'includeConditionCodes'`	Exchange condition codes associated with the event
`'includeExchangeCodes'`	Exchange code where tick originated
`'includeBrokerCodes'`	Broker code
`'includeRpsCodes'`	Reporting party side
`'includeNonPlottableEvents'`	After-hours data

Note The value `'includeNonPlottableEvents'` applies to raw intraday requests only.

To specify more than one Bloomberg API option, use a cell array of these values.

Specify the corresponding Bloomberg API value for each API option. The number of options must match the number of values.

For example, to specify one Bloomberg API option, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               'includeConditionCodes','true');
```

To specify two Bloomberg API options, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               {'includeConditionCodes','includeExchangeCodes'},...
               {'true','true'});
```

For details about the options, see the *Bloomberg API Developer's Guide*.

Data Types: `char` | `cell`

`values` — Bloomberg API values
`'true'` | `'false'`

Bloomberg API values, specified as `'true'` or `'false'`. Each value corresponds to the specified Bloomberg API option. To specify more than one Bloomberg API value, use a cell array. The number of values must match the number of options.

For example, to specify one Bloomberg API option, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               'includeConditionCodes','true');
```

To specify two Bloomberg API options, enter:

```
d = timeseries(c,'F US Equity',floor(now),[],'Trade',...
               {'includeConditionCodes','includeExchangeCodes'},...
               {'true','true'});
```

Data Types: `char` | `cell`

`startdate` — Start date
numeric scalar | character vector | string scalar | `datetime` array

Start date, specified as a numeric scalar, character vector, string scalar, or `datetime` array. This date specifies the beginning of the date range for the returned tick data. If no ticks are present in the date range, then returned tick data is empty.

Example: `floor(now-1)`

Data Types: `double` | `char` | `string` | `datetime`

`enddate` — End date
numeric scalar | character vector | string scalar | `datetime` array

End date, specified as a numeric scalar, character vector, string scalar, or `datetime` array. This date specifies the end of the date range for the returned tick data. If no ticks are present in the date range, then returned tick data is empty.

Example: `floor(now)`

Data Types: `double` | `char` | `string` | `datetime`

`starttime` — Start time
character vector | string scalar | `datetime` array

Start time, specified as a character vector, string scalar, or `datetime` array. This time specifies the start time of the time range for the returned tick data.

Example: `'09:30:00'`

Data Types: `char` | `string` | `datetime`

`endtime` — End time
character vector | string scalar | `datetime` array

End time, specified as a character vector, string scalar, or `datetime` array. This time specifies the end time of the time range for the returned tick data.

Example: `'16:30:00'`

Data Types: `char` | `string` | `datetime`

`dayincrement` — Day increment
1 (default) | numeric scalar

Day increment, specified as a numeric scalar. This number specifies the whole day increment for a specific date range. For example, if the day increment is 7, then the returned data contains ticks for every 7th day starting from the first day within the date range.

Data Types: `double`

Output Arguments

`d` — Bloomberg tick data
cell array | numeric array | table | timetable

Bloomberg tick data, returned as one of these data types:

- Cell array for requests without a specified time interval (raw tick data)
- Numeric array for requests with a specified time interval
- table
- timetable

The data type of the tick data depends on the DataReturnFormat and DatetimeType properties of the connection object.

Note The Bloomberg API returns the tick time with precision in seconds.

Limitations

When the data request is too large, `timeseries` displays this error message:

```
Timeout error:
Error using blp/timeseries>processResponseEvent (line 338) REQUEST FAILED: responseError = {

source = bbdbl7

code = -2

category = TIMEOUT

message = Timed out getting data from store [nid:327]
```

```
subcategory = INTERNAL_ERROR
}
```

To fix this error, shorten the length of the date range by modifying the input arguments `startdate` and `enddate`.

Tips

- You cannot retrieve Bloomberg intraday tick data for a date more than 140 days ago.
- The *Bloomberg API Developer's Guide* states that `'TRADE'` corresponds to LAST_PRICE for IntradayTickRequest and IntradayBarRequest.
- Bloomberg V3 intraday tick data supports additional name-value pairs. For details on these pairs, see the *Bloomberg API Developer's Guide* by typing `WAPI` and clicking the **<GO>** button on the Bloomberg terminal.
- You can check data and field availability by using the Bloomberg Excel Add-In.

Version History

Introduced in R2021a

See Also

`bloombergServer` | `close` | `getdata` | `history` | `realtime`

Topics

"Retrieve Bloomberg Intraday Tick Data Using Bloomberg Server C++ Interface" on page 5-45

bloombergEMSX

Create Bloomberg EMSX connection

Description

The `bloombergEMSX` function creates a `bloombergEMSX` object, which represents a Bloomberg EMSX connection using the Bloomberg V3 C++ API. After you create an `bloombergEMSX` object, you can use the object functions to create and route orders, and then manage orders and routes. For details about Bloomberg EMSX, see the *EMSX API Programmers Guide*.

Creation

Syntax

```
c = bloombergEMSX(servicename)

c = bloombergEMSX(servicename,authid,serverip)
c = bloombergEMSX(servicename,authid,serverip,portnumber)
c = bloombergEMSX(servicename,authid,serverip,portnumber,terminalip)
```

Description

Local Connection

`c = bloombergEMSX(servicename)` creates a connection to the local Bloomberg EMSX communications server using the service `servicename` with the Bloomberg EMSX C++ interface.

Remote Connection

`c = bloombergEMSX(servicename,authid,serverip)` creates a connection to a remote EMSX server using the specified service name, authentication identifier, and server IP address.

`c = bloombergEMSX(servicename,authid,serverip,portnumber)` also specifies the port number of the machine running the EMSX Server process.

`c = bloombergEMSX(servicename,authid,serverip,portnumber,terminalip)` also specifies the IP address of the machine you use to access the Bloomberg Terminal for the remote connection.

Input Arguments

`servicename` — Bloomberg EMSX service name
`'//blp/emapisvc_beta'` | `'//blp/emapisvc'`

Bloomberg EMSX service name, specified as one of these connection types.

Connection Type	Bloomberg EMSX Service Name
Test	`'//blp/emapisvc_beta'`

Connection Type	Bloomberg EMSX Service Name
Production	'//blp/emapisvc'

`authid` — Bloomberg EMSX authentication identifier
character vector | string scalar

Bloomberg EMSX authentication identifier, specified as a character vector or string scalar.

This input argument is required for Bloomberg EMSX Server. If you are using Bloomberg EMSX Desktop, specify an empty character vector or string scalar because this input argument is not required.

`serverip` — Bloomberg EMSX Server IP address
character vector | string scalar

Bloomberg EMSX Server IP address, specified as a character vector or string scalar. This address is the IP address of the machine running the Bloomberg EMSX Server process.

This input argument is required for Bloomberg EMSX Server. If you are using Bloomberg EMSX Desktop, specify an empty character vector or string scalar because this input argument is not required.

Example: `'111.222.333.44'`

`portnumber` — Port number
`8194` (default) | numeric scalar

Port number of the machine running the EMSX Server process, specified as a numeric scalar.

This input argument is required for Bloomberg EMSX Server. If you are using Bloomberg EMSX Desktop, specify an empty array because this input argument is not required.

`terminalip` — Bloomberg Terminal IP address
`"localhost"` (default) | character vector | string scalar

Bloomberg Terminal IP address, specified as a character vector or string scalar. This address is the IP address of the machine you use to access the Bloomberg Terminal.

Example: `'111.222.333.44'`

Properties

`Session` — Bloomberg EMSX session
session object

This property is read-only.

Bloomberg EMSX session, specified as a Bloomberg EMSX session object.

Example: `[1x1 datafeed.internal.BLPSession]`

`Service` — Bloomberg EMSX service
character vector

This property is read-only.

Bloomberg EMSX service, specified as a character vector.

The `bloombergEMSX` function sets this property using the `servicename` input argument.

Example: `'//blp/emapisvc_beta'`

`Ipaddress` — IP address
`'localhost'` (default) | character vector

This property is read-only.

IP address of the machine running Bloomberg EMSX, specified as a character vector.

Data Types: `char`

`Port` — Port number
numeric scalar

This property is read-only.

Port number of the machine running Bloomberg EMSX, specified as a numeric scalar.

Example: `8194`

Data Types: `double`

`User` — User
Bloomberg API C++ object

This property is read-only.

User, specified as a Bloomberg API C++ object for Bloomberg EMSX Server. For Bloomberg EMSX Desktop, this property is empty.

Example: `[1×1 com.bloomberglp.blpapi.impl.by]`

Object Functions

Create Bloomberg EMSX Connection

orders	Obtain Bloomberg EMSX order subscription
routes	Obtain Bloomberg EMSX route subscription
close	Close Bloomberg EMSX connection

Create Bloomberg EMSX Orders and Routes

createOrder	Create Bloomberg EMSX order
routeOrder	Route Bloomberg EMSX order
routeOrderWithStrat	Route Bloomberg EMSX order with strategies
groupRouteOrder	Route group of Bloomberg EMSX orders
groupRouteOrderWithStrat	Route group of Bloomberg EMSX orders with strategies
createOrderAndRoute	Create and route Bloomberg EMSX order
createOrderAndRouteWithStrat	Create and route Bloomberg EMSX order with strategies
createBasket	Create basket of Bloomberg EMSX orders

Manage Bloomberg EMSX Orders and Routes

manualFill	Fill Bloomberg EMSX orders manually
modifyOrder	Modify Bloomberg EMSX order
modifyRoute	Modify Bloomberg EMSX route
modifyRouteWithStrat	Modify Bloomberg EMSX route with strategies
deleteOrder	Delete Bloomberg EMSX order
deleteRoute	Delete Bloomberg EMSX active shares
processEvent	Sample Bloomberg EMSX event handler

Retrieve Bloomberg EMSX Information

getBrokerInfo	Obtain Bloomberg EMSX broker and strategy information
getAllFieldMetaData	Obtain Bloomberg EMSX field information

Examples

Connect to Bloomberg EMSX Test Service

First, create a Bloomberg EMSX test service connection. Then, obtain broker information.

Create a connection `c` to the Bloomberg EMSX test service using the Bloomberg EMSX C++ interface. You can place test calls using this service.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

```
c =

  bloombergEMSX with properties:

    Session: [1×1 datafeed.internal.BLPSession]
    Service: '//blp/emapisvc_beta'
    Ipaddress: "111.222.333.44"
    Port: 8194.00
    User: []
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following properties:

- Bloomberg EMSX session object
- Bloomberg EMSX service
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service
- Bloomberg API C++ object

Define the broker and strategy information structure `brokerstrat`. Obtain broker information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

The `EMSX_BROKERS` field lists the Bloomberg EMSX brokers.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
```

```
r = getBrokerInfo(c,brokerstrat)
```

```
 r =

        EMSX_BROKERS: {2x1 cell}
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Connect to Bloomberg EMSX Production Service

First, create a Bloomberg EMSX production service connection. Then, obtain broker information.

Create a connection `c` to the Bloomberg EMSX production service using the Bloomberg EMSX C++ interface. You can place live calls using this service.

```
c = bloombergEMSX('//blp/emapisvc')
```

```
c =

  bloombergEMSX with properties:

    Session: [1×1 datafeed.internal.BLPSession]
    Service: '//blp/emapisvc'
    Ipaddress: "111.222.333.44"
    Port: 8194.00
    User: []
```

MATLAB returns `c` as the connection to the Bloomberg EMSX production service with the following properties:

- Bloomberg EMSX session object
- Bloomberg EMSX service
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service
- Bloomberg API C++ object

Define the broker and strategy information structure `brokerstrat`. Obtain broker information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

The `EMSX_BROKERS` field lists the Bloomberg EMSX brokers.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
```

```
r = getBrokerInfo(c,brokerstrat)
```

```
 r =

        EMSX_BROKERS: {2x1 cell}
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Connect to Bloomberg EMSX Remote Server

Obtain broker information using a Bloomberg EMSX test connection to a remote server.

Create a connection `c` to the Bloomberg EMSX remote server using the Bloomberg EMSX C++ interface. Specify the service name, authentication identifier, and server IP address.

```
servicename = '//blp/emapisvc_beta';
authid = 'abcdef123';
serverip = '111.222.333.44';
c = bloombergEMSX(servicename,authid,serverip)
```

```
c =

  bloombergEMSX with properties:

    Session: [1×1 datafeed.internal.BLPSession]
    Service: '//blp/emapisvc_beta'
    Ipaddress: "111.222.333.44"
    Port: 8194.00
    User: []
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following properties:

- Bloomberg EMSX session object
- Bloomberg EMSX service
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service
- Bloomberg API C++ object

Define the broker and strategy information structure `brokerstrat`. Obtain broker information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

The `EMSX_BROKERS` field lists the Bloomberg EMSX brokers.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
```

```
r = getBrokerInfo(c,brokerstrat)
```

```
 r =

        EMSX_BROKERS: {2x1 cell}
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Connect to Bloomberg EMSX Remote Server with Port Number

Obtain broker information using a Bloomberg EMSX test connection to a remote server with a port number.

Create a connection `c` to the Bloomberg EMSX remote server using the Bloomberg EMSX C++ interface. Specify the service name, authentication identifier, server IP address, and port number.

```
servicename = '//blp/emapisvc_beta';
authid = 'abcdef123';
serverip = '111.222.333.44';
portnumber = 5678;
c = bloombergEMSX(servicename,authid,serverip,portnumber)

c =

  bloombergEMSX with properties:

    Session: [1×1 datafeed.internal.BLPSession]
    Service: '//blp/emapisvc_beta'
    Ipaddress: "111.222.333.44"
    Port: 5678.00
    User: []
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following properties:

- Bloomberg EMSX session object
- Bloomberg EMSX service
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service
- Bloomberg API C++ object

Define the broker and strategy information structure `brokerstrat`. Obtain broker information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

The `EMSX_BROKERS` field lists the Bloomberg EMSX brokers.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';

r = getBrokerInfo(c,brokerstrat)

 r =

        EMSX_BROKERS: {2x1 cell}
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Connect to Bloomberg EMSX Remote Server with Terminal IP Address

Obtain broker information using a Bloomberg EMSX test connection to a remote server with a port number and Bloomberg Terminal IP address.

Create a connection `c` to the Bloomberg EMSX remote server using the Bloomberg EMSX C++ interface. Specify the service name, authentication identifier, server IP address, and port number. Also, specify the IP address of the machine you use to access the Bloomberg Terminal.

```
servicename = '//blp/emapisvc_beta';
authid = 'abcdef123';
serverip = '111.222.333.44';
```

```
portnumber = 8194;
terminalip = '5555.222.333.44';
c = bloombergEMSX(servicename,authid,serverip,portnumber,terminalip)

c =

  bloombergEMSX with properties:

    Session: [1×1 datafeed.internal.BLPSession]
    Service: '//blp/emapisvc_beta'
    Ipaddress: "111.222.333.44"
    Port: 8194.00
    User: []
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following properties:

- Bloomberg EMSX session object
- Bloomberg EMSX service
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service
- Bloomberg API C++ object

Define the broker and strategy information structure `brokerstrat`. Obtain broker information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

The `EMSX_BROKERS` field lists the Bloomberg EMSX brokers.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';

r = getBrokerInfo(c,brokerstrat)

 r =

        EMSX_BROKERS: {2x1 cell}
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Version History

Introduced in R2021a

See Also

Topics

“Create Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-2
“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11

External Websites
EMSX API Programmers Guide

close

Close Bloomberg EMSX connection

Syntax

```
close(c)
```

Description

`close(c)` closes the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

Examples

Close the Bloomberg EMSX Connection

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `createOrder` | `createOrderAndRoute` | `routeOrder`

Topics

“Create Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-2
“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11

External Websites

EMSX API Programmers Guide

createBasket

Create basket of Bloomberg EMSX orders

Syntax

```
events = createBasket(c,basket,order)
events = createBasket(c,basket,'timeOut',timeout)
createBasket( ___ ,'useDefaultEventHandler',false)
 ___  = createBasket(c,basket,options)
```

Description

`events = createBasket(c,basket,order)` creates a basket of Bloomberg EMSX orders using the Bloomberg EMSX connection with the Bloomberg EMSX C++ interface, basket name, and order request. `createBasket` returns the order sequence numbers and status message using the default event handler.

`events = createBasket(c,basket,'timeOut',timeout)` specifies a timeout value for the execution of the default event handler.

`createBasket( ___ ,'useDefaultEventHandler',false)` creates a basket of Bloomberg EMSX orders using any of the previous input argument combinations and a custom event handler function. Write a custom event handler to process the events associated with creating a basket of orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue.

`___  = createBasket(c,basket,options)` uses the `options` structure to customize the output, which is useful for configuring and saving your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the `useDefaultEventHandler` field is set to `true`, and omit this output argument when the `useDefaultEventHandler` field is set to `false`.

Examples

Create Basket of Bloomberg EMSX Orders Using Default Event Handler

Using a Bloomberg EMSX connection, create a basket of Bloomberg EMSX orders.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
```

```
order1.EMSX_ORDER_TYPE = 'MKT';
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Create a basket of the two existing orders. Specify the basket name. Then, specify the order numbers in the `orders` structure.

```
basket = 'OrderBasket';
orders.EMSX_SEQUENCE = [int32(354646);int32(354777)];
events = createBasket(c,basket,orders)
```

```
events =

  struct with fields:
```

```
    EMSX_SEQUENCE: [2×1 double]
    MESSAGE: 'Orders added to Basket'
```

The default event handler processes the events associated with creating a basket of orders. `events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order numbers
- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Create Basket of Bloomberg EMSX Orders Using Timeout Value

Using a Bloomberg EMSX connection, create a basket of Bloomberg EMSX orders. Specify a timeout value.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
order1.EMSX_ORDER_TYPE = 'MKT';
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force

set to DAY and any hand instruction. Convert the number of shares to a 32-bit signed integer using int32.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection c and order2.

```
events = createOrder(c,order2)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Create a basket of the two existing orders. Specify the basket name. Then, specify the order numbers as the structure orders. Specify the timeout value of 200 milliseconds by using the 'timeOut' flag.

```
basket = 'OrderBasket';
orders.EMSX_SEQUENCE = [int32(354646);int32(354777)];
events = createBasket(c,basket,orders,'timeOut',200)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: [2×1 double]
    MESSAGE: 'Orders added to Basket'
```

events is a structure that contains these fields:

- EMSX_SEQUENCE — Bloomberg EMSX order numbers
- MESSAGE — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Create Basket of Bloomberg EMSX Orders Using Custom Event Handler

Using a Bloomberg EMSX connection, create a basket of Bloomberg EMSX orders. Specify using a custom event handler function to process the events.

To create a Bloomberg EMSX order, create the connection c using emsx and set up the order subscription using orders. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure order1 to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker BB with the time in force

set to DAY and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
order1.EMSX_ORDER_TYPE = 'MKT';
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Create a basket of the two existing orders. Specify the basket name. Then, specify the order numbers as the structure `orders`. Use a custom event handler function to process the events. You can use the sample event handler function `processEvent` or write your own custom event handler function. For this example, use `processEvent` to process the events.

```
basket = 'OrderBasket';
orders.EMSX_SEQUENCE = [int32(354646);int32(354777)];
```

```
createBasket(c,basket,orders,'useDefaultEventHandler',false)
processEvent(c)

CreateBasket = {

    EMSX_SEQUENCE[] = {

        354646, 354777

    }

    MESSAGE = 'Orders added to Basket'

}
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create Basket of Bloomberg EMSX Orders Using Options Structure

Using a Bloomberg EMSX connection, create a basket of Bloomberg EMSX orders. Specify an additional option for a timeout value by using a structure.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
order1.EMSX_ORDER_TYPE = 'MKT';
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)

events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Create a basket of the two existing orders. Specify the basket name. Then, specify the order numbers as the structure `orders`. Specify an additional option for a timeout value of 200 milliseconds by using the `options` structure.

```
basket = 'OrderBasket';
orders.EMSX_SEQUENCE = [int32(354646);int32(354777)];
options.timeOut = 200;
events = createBasket(c,basket,orders,options)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: [2×1 double]
    MESSAGE: 'Orders added to Basket'
```

`events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order numbers
- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

`basket` — Basket name
character vector | string scalar

Basket name, specified as a character vector or string scalar.

Example: `"OrderBasket"`

Data Types: `char` | `string`

`order` — Order request
structure

Order request, specified as a structure that contains the `EMSX_SEQUENCE` field. This field contains the order numbers. Convert the order numbers to a 32-bit signed integer by using `int32`.

Example: `int32(123456)`

Data Types: `struct`

`timeout` — Timeout value
`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `routes` | `close` | `createOrder` | `routeOrder` | `groupRouteOrder` | `modifyOrder` | `deleteOrder` | `processEvent`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

createOrder

Create Bloomberg EMSX order

Syntax

```
events = createOrder(c,order)
events = createOrder(c,order,'timeOut',timeout)

createOrder( ___ ,'useDefaultEventHandler',false)

 ___  = createOrder(c,order,options)
```

Description

`events = createOrder(c,order)` creates a Bloomberg EMSX order using the Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface and order request `order` that contains the required fields for creating an order. `createOrder` returns the order sequence number and status message using the default event handler.

`events = createOrder(c,order,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`createOrder( ___ ,'useDefaultEventHandler',false)` creates a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with creating orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

` ___  = createOrder(c,order,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Create an Order Using the Default Event Handler

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
```

```
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order`.

```
events = createOrder(c,order)
```

```
events =

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create an Order Using a Timeout

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order`. Set the timeout value to 200 milliseconds.

```
events = createOrder(c,order,'timeOut',200)
```

```
events =
```

```
    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create an Order Using a Custom Event Handler

To create a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Create the order using the Bloomberg EMSX connection `c` and `order`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with creating an order.

```
createOrder(c,order,'useDefaultEventHandler',false)
```

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(subs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create an Order Using an Options Structure

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Create the order using the Bloomberg EMSX connection `c`, `order`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = createOrder(c,order,options)
```

```
events =

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

order — Order request

structure

Order request, specified as a structure using Bloomberg EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `order`. Convert the number of shares to a 32-bit signed integer using `int32`. `order` contains these fields.

Field	Description
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX amount of shares
`EMSX_ORDER_TYPE`	Bloomberg EMSX order type
`EMSX_BROKER`	Bloomberg EMSX broker name
`EMSX_TIF`	Bloomberg EMSX time in force
`EMSX_HAND_INSTRUCTION`	Bloomberg EMSX hand instruction
`EMSX_SIDE`	Bloomberg EMSX buy or sell specification

Example:
```
order.EMSX_TICKER = 'XYZ';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Data Types: `struct`

timeout — Timeout value

`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

options — Options for custom event handler or timeout value

structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `routes` | `close` | `routeOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `manualFill` | `modifyOrder` | `deleteOrder` | `deleteRoute` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

createOrderAndRoute

Create and route Bloomberg EMSX order

Syntax

```
events = createOrderAndRoute(c,order)
events = createOrderAndRoute(c,order,'timeOut',timeout)

createOrderAndRoute(___,'useDefaultEventHandler',false)

___ = createOrderAndRoute(c,order,options)
```

Description

`events = createOrderAndRoute(c,order)` creates and routes a Bloomberg EMSX order using Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface and order request `order`. `createOrderAndRoute` returns the order sequence number, route number, and status message using the default event handler.

`events = createOrderAndRoute(c,order,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`createOrderAndRoute(___,'useDefaultEventHandler',false)` creates and routes a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with creating and routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___ = createOrderAndRoute(c,order,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Create and Route an Order Using the Default Event Handler

To create and route a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
```

```
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection `c` and `order`.

```
events = createOrderAndRoute(c,order)
```

```
events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using a Timeout

To create and route a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection `c` and `order`. Set the timeout value to 200 milliseconds.

```
events = createOrderAndRoute(c,order,'timeOut',200)
```

```
events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using a Custom Event Handler

To create and route a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Create and route the order using the Bloomberg EMSX connection `c` and `order`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with creating and routing an order.

```
createOrderAndRoute(c,order,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using an Options Structure

To create and route a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Create and route the order using the Bloomberg EMSX connection `c`, `order`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = createOrderAndRoute(c,order,options)
```

```
events =

    EMSX_SEQUENCE: 728924
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number

- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

`order` — Order request

structure

Order request, specified as a structure using Bloomberg EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `order`. Convert the number of shares to a 32-bit signed integer using `int32`. `order` contains these fields.

Field	Description
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX amount of shares
`EMSX_ORDER_TYPE`	Bloomberg EMSX order type
`EMSX_BROKER`	Bloomberg EMSX broker name
`EMSX_TIF`	Bloomberg EMSX time in force
`EMSX_HAND_INSTRUCTION`	Bloomberg EMSX hand instruction
`EMSX_SIDE`	Bloomberg EMSX buy or sell specification

Example: `order.EMSX_TICKER = 'XYZ';`

```
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Data Types: `struct`

`timeout` — Timeout value

`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `routes` | `close` | `createOrder` | `routeOrder` | `createOrderAndRouteWithStrat` | `modifyOrder` | `deleteOrder` | `deleteRoute` | `timer` | `start` | `stop` | `delete`

Topics

“Create Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-2
“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

createOrderAndRouteWithStrat

Create and route Bloomberg EMSX order with strategies

Syntax

```
events = createOrderAndRouteWithStrat(c,order,strat)
events = createOrderAndRouteWithStrat(c,order,strat,'timeOut',timeout)

createOrderAndRouteWithStrat( ___ ,'useDefaultEventHandler',false)

 ___  = createOrderAndRouteWithStrat(c,order,strat,options)
```

Description

`events = createOrderAndRouteWithStrat(c,order,strat)` creates and routes a Bloomberg EMSX order with strategies using Bloomberg EMSX connection `c` and the Bloomberg EMSX C++ interface, order request `order`, and order strategy `strat`. `createOrderAndRouteWithStrat` returns the order sequence number, route number, and status message using the default event handler.

`events = createOrderAndRouteWithStrat(c,order,strat,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`createOrderAndRouteWithStrat( ___ ,'useDefaultEventHandler',false)` creates and routes a Bloomberg EMSX order with strategies using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with creating and routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

` ___  = createOrderAndRouteWithStrat(c,order,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Create and Route an Order Using the Default Event Handler

To create and route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create and route the order with strategies using the Bloomberg EMSX connection `c`, `order`, and `strat`.

```
events = createOrderAndRouteWithStrat(c,order,strat)
```

```
events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using a Timeout

To create and route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
```

```
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create and route the order with strategies using the Bloomberg EMSX connection `c`, `order`, and `strat`. Set the timeout value to 200 milliseconds.

```
events = createOrderAndRouteWithStrat(c,order,strat,'timeOut',200)
```

```
events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using a Custom Event Handler

To create and route a Bloomberg EMSX order with strategies, create the Bloomberg EMSX connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
```

```
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Create and route the order with strategies using the Bloomberg EMSX connection `c`, `order`, and `strat`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with creating and routing an order.

```
createOrderAndRouteWithStrat(c,order,strat,...
                             'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using an Options Structure

To create and route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Create and route the order using the Bloomberg EMSX connection `c`, `order`, `strat`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = createOrderAndRouteWithStrat(c,order,strat,options)
```

```
events =

    EMSX_SEQUENCE: 728924
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

`order` — Order request

structure

Order request, specified as a structure using Bloomberg EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `order`. Convert the number of shares to a 32-bit signed integer using `int32`. `order` contains these fields.

Field	Description
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX amount of shares
`EMSX_ORDER_TYPE`	Bloomberg EMSX order type
`EMSX_BROKER`	Bloomberg EMSX broker name
`EMSX_TIF`	Bloomberg EMSX time in force
`EMSX_HAND_INSTRUCTION`	Bloomberg EMSX hand instruction
`EMSX_SIDE`	Bloomberg EMSX buy or sell specification

Example:
```
order.EMSX_TICKER = 'XYZ';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Data Types: `struct`

`strat` — Order strategies
structure

Order strategies, specified as a structure that contains the fields: `EMSX_STRATEGY_NAME`, `EMSX_STRATEGY_FIELD_INDICATORS`, and `EMSX_STRATEGY_FIELDS`. The structure field values must align with the strategy fields specified by `EMSX_STRATEGY_NAME`. For details about strategy fields and ordering, see `getBrokerInfo`.

Convert `EMSX_STRATEGY_FIELD_INDICATORS` to a 32-bit signed integer using `int32`. Set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `0` for each field to use the field data setting in `EMSX_FIELD_DATA`. Or, set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `1` to ignore the data in `EMSX_FIELD_DATA`.

Example:
```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Data Types: `struct`

`timeout` — Timeout value
`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `routes` | `close` | `createOrder` | `routeOrder` | `modifyOrder` | `deleteOrder` | `deleteRoute` | `getBrokerInfo` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

deleteOrder

Delete Bloomberg EMSX order

Syntax

```
events = deleteOrder(c,ordernum)
events = deleteOrder(c,ordernum,'timeOut',timeout)

deleteOrder( ___ ,'useDefaultEventHandler',false)

 ___  = deleteOrder(c,ordernum,options)
```

Description

`events = deleteOrder(c,ordernum)` deletes a Bloomberg EMSX order using the Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface and order number or structure `ordernum`. `deleteOrder` returns a status message using the default event handler.

`events = deleteOrder(c,ordernum,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`deleteOrder( ___ ,'useDefaultEventHandler',false)` deletes a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with deleting orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___  = deleteOrder(c,ordernum,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Delete an Order Using the Default Event Handler

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Delete the order using the Bloomberg EMSX connection `c` and `ordernum`.

```
events = deleteOrder(c,ordernum)
```

```
events =

      STATUS: '0'
     MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting the order. `deleteOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete an Order Using the Order Number Integer

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Delete the order using the Bloomberg EMSX connection `c` and the order sequence number `335877` for the order to delete.

```
events = deleteOrder(c,335877)
```

```
events =

      STATUS: '0'
     MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting the order. `deleteOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete an Order Using a Timeout

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Delete the order using the Bloomberg EMSX connection `c` and `ordernum`. Set the timeout value to 200 milliseconds.

```
events = deleteOrder(c,ordernum,'timeOut',200)
```

```
events =

     STATUS: '0'
    MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting the order. `deleteOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete an Order Using a Custom Event Handler

To delete a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Delete the order using the Bloomberg EMSX connection `c` and `ordernum`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with deleting an order.

```
deleteOrder(c,ordernum,'useDefaultEventHandler',false)
```

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(subs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete an Order Using an Options Structure

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Delete the order using the Bloomberg EMSX connection `c`, `ordernum`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = deleteOrder(c,ordernum,options)
```

```
events =

     STATUS: '0'
    MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting the order. `deleteOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

`ordernum` — Order numbers to delete

structure | integer

Order numbers to delete, specified as a structure or an integer to denote one or more order sequence numbers.

Data Types: `struct` | `int32`

`timeout` — Timeout value

`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value

structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `routes` | `close` | `createOrder` | `routeOrder` | `createOrderAndRoute` | `modifyOrder` | `deleteRoute` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

deleteRoute

Delete Bloomberg EMSX active shares

Syntax

```
events = deleteRoute(c,routenum)
events = deleteRoute(c,routenum,'timeOut',timeout)

deleteRoute( ___ ,'useDefaultEventHandler',false)

 ___  = deleteRoute(c,routenum,options)
```

Description

`events = deleteRoute(c,routenum)` deletes the active shares that are routed but not filled using the Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface and route number `routenum`. `deleteRoute` returns a status message using the default event handler.

`events = deleteRoute(c,routenum,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`deleteRoute( ___ ,'useDefaultEventHandler',false)` deletes the active shares that are routed but not filled using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with deleting the active shares. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___  = deleteRoute(c,routenum,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Delete Active Shares

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

1. Create the connection `c` using `emsx`.
2. Set up an order and route subscription using `orders` and `routes`.
3. Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see “Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7.

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and `routenum`.

```
events = deleteRoute(c,routenum)
```

```
events =

     STATUS: '1'
    MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting the active shares. `deleteRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete Active Shares Using a Timeout

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

1. Create the connection `c` using `emsx`.
2. Set up an order and route subscription using `orders` and `routes`.
3. Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see “Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7.

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and `routenum`. Set the timeout value to 200 milliseconds.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;
```

```
events = deleteRoute(c,routenum,'timeOut',200)

events =

     STATUS: '1'
    MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting the active shares. `deleteRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete Active Shares Using a Custom Event Handler

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

1. Create the Bloomberg EMSX connection `c` using `emsx`.
2. Set up an order and route subscription using `orders` and `routes`.
3. Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see “Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7.

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and `routenum`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with deleting the active shares.

```
deleteRoute(c,routenum,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete Active Shares Using an Options Structure

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

1. Create the connection `c` using `emsx`.
2. Set up an order and route subscription using `orders` and `routes`.
3. Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see “Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7.

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c`, `routenum`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = deleteRoute(c,routenum,options)
```

```
events =

     STATUS: '1'
    MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting the active shares. `deleteRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status

- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

`routenum` — Route to delete

structure

Route to delete, specified as a structure containing fields `EMSX_SEQUENCE` and `EMSX_ROUTE_ID`.

Example: `routenum.EMSX_SEQUENCE = 728918;`
`routenum.EMSX_ROUTE_ID = 1;`

Data Types: `struct`

`timeout` — Timeout value

`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value

structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `routes` | `close` | `createOrder` | `routeOrder` | `createOrderAndRoute` | `modifyOrder` | `modifyRoute` | `deleteOrder` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

getAllFieldMetaData

Obtain Bloomberg EMSX field information

Syntax

```
r = getAllFieldMetaData(c)
```

Description

`r = getAllFieldMetaData(c)` returns the Bloomberg EMSX field information using the Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface.

Examples

Request All Field Information

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Request all fields supported by Bloomberg EMSX service using the Bloomberg EMSX connection `c`.

```
r = getAllFieldMetaData(c)
```

```
r =

       EMSX_FIELD_NAME: {113x1 cell}
        EMSX_DISP_NAME: {113x1 cell}
             EMSX_TYPE: {113x1 cell}
            EMSX_LEVEL: [113x1 double]
              EMSX_LEN: [113x1 double]
```

Display all field information for the first Bloomberg EMSX field using a cell array. Create a cell array from the fields in the returned data structure `r`.

```
{r.EMSX_FIELD_NAME{1} r.EMSX_DISP_NAME{1} r.EMSX_TYPE{1} r.EMSX_LEVEL(1) r.EMSX_LEN(1)}
```

```
 'MSG_TYPE'    'Msg Type'    'String'    [0]    [1]
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

Output Arguments

`r` — Return information for all fields
structure

Return information for all fields, returned as a structure for all fields supported by Bloomberg EMSX.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `close` | `createOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11

External Websites

EMSX API Programmers Guide

getBrokerInfo

Obtain Bloomberg EMSX broker and strategy information

Syntax

```
r = getBrokerInfo(c,brokerstrat)
```

Description

`r = getBrokerInfo(c,brokerstrat)` obtains Bloomberg EMSX broker and strategy information using the Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface and broker and strategy request structure `brokerstrat`.

Examples

Obtain Broker Information

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Define the broker and strategy information structure `brokerstrat`. Obtain broker information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
```

```
r = getBrokerInfo(c,brokerstrat)
```

```
 r =

        EMSX_BROKERS: {2x1 cell}
```

The `EMSX_BROKERS` field lists the Bloomberg EMSX brokers.

Close the Bloomberg EMSX connection.

```
close(c)
```

Obtain Strategy Information

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Define the broker and strategy information structure `brokerstrat`. Obtain strategy information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
brokerstrat.EMSX_BROKER = 'BMTB';
```

```
r = getBrokerInfo(c,brokerstrat)

 r =

        EMSX_STRATEGIES: {16x1 cell}
```

The `EMSX_STRATEGIES` field lists the Bloomberg EMSX strategies.

Close the Bloomberg EMSX connection.

```
close(c)
```

Obtain Field Information

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Define the broker and strategy information structure `brokerstrat`. Obtain field information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
brokerstrat.EMSX_BROKER = 'BMTB';
brokerstrat.EMSX_STRATEGY = 'SSP';

r = getBrokerInfo(c,brokerstrat)

r =

        FieldName: {3x1 cell}
        Disable: {3x1 cell}
        StringValue: {3x1 cell}
```

The structure field `FieldName` lists the Bloomberg EMSX fields. The structure fields `Disable` and `StringValue` contain information about the Bloomberg EMSX fields.

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

`brokerstrat` — Broker and strategy request

structure

Broker and strategy request, specified as a structure that contains Bloomberg EMSX fields. Use `getAllFieldMetaData` to view all available fields for `brokerStrategyStruct`.

Example: `brokerstrat.EMSX_TICKER = 'ABCD US Equity';`

Data Types: `struct`

Output Arguments

`r` — Broker and strategy information

structure

Broker and strategy information, returned as a structure.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `routes` | `close` | `createOrder` | `routeOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `modifyOrder` | `deleteOrder` | `deleteRoute`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11

External Websites

EMSX API Programmers Guide

groupRouteOrder

Route group of Bloomberg EMSX orders

Syntax

```
events = groupRouteOrder(c,order)
events = groupRouteOrder(c,order,'timeOut',timeout)
groupRouteOrder( ___ ,'useDefaultEventHandler',false)
 ___  = groupRouteOrder(c,order,options)
```

Description

`events = groupRouteOrder(c,order)` routes a group of Bloomberg EMSX orders using the Bloomberg EMSX connection with the Bloomberg EMSX C++ interface and order request. `groupRouteOrder` returns the order sequence number and status message using the default event handler.

`events = groupRouteOrder(c,order,'timeOut',timeout)` specifies a timeout value for the execution of the default event handler.

`groupRouteOrder( ___ ,'useDefaultEventHandler',false)` routes a group of Bloomberg EMSX orders using any of the previous input argument combinations and a custom event handler function. Write a custom event handler to process the events associated with routing a group of orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue.

`___  = groupRouteOrder(c,order,options)` uses the `options` structure to customize the output, which is useful for configuring and saving your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the `useDefaultEventHandler` field is set to `true`, and omit this output argument when the `useDefaultEventHandler` field is set to `false`.

Examples

Route Group of Bloomberg EMSX Orders Using Default Event Handler

Using a Bloomberg EMSX connection, route a group of Bloomberg EMSX orders.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
```

```
order1.EMSX_ORDER_TYPE = 'MKT';
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Route the two existing orders. Specify the order numbers, broker, and hand instruction in the `order` structure.

```
order.EMSX_SEQUENCE{1} = int32(354646);
order.EMSX_SEQUENCE{2} = int32(354777);
order.EMSX_BROKER = 'BB';
order.EMSX_HAND_INSTRUCTION = 'ANY';
events = groupRouteOrder(c,order)
```

```
events =
```

```
  struct with fields:

    EMSX_SEQUENCE: 354646
    EMSX_ROUTE_ID: 1
    MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing a group of orders. `events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order numbers
- `EMSX_ROUTE_ID` — Bloomberg EMSX route identifier
- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Group of Bloomberg EMSX Orders Using Timeout Value

Using a Bloomberg EMSX connection, route a group of Bloomberg EMSX orders. Specify a timeout value.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
order1.EMSX_ORDER_TYPE = 'MKT';
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number

- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Route the two existing orders. Specify the order numbers, broker, and hand instruction in the `order` structure. Specify an additional option for a timeout value of 200 milliseconds by using the `'timeOut'` flag.

```
order.EMSX_SEQUENCE{1} = int32(354646);
order.EMSX_SEQUENCE{2} = int32(354777);
order.EMSX_BROKER = 'BB';
order.EMSX_HAND_INSTRUCTION = 'ANY';
events = groupRouteOrder(c,order,'timeOut',200)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    EMSX_ROUTE_ID: 1
    MESSAGE: 'Order Routed'
```

`events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order numbers
- `EMSX_ROUTE_ID` — Bloomberg EMSX route identifier
- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Group of Bloomberg EMSX Orders Using Custom Event Handler Function

Using a Bloomberg EMSX connection, route a group of Bloomberg EMSX orders. Specify using a custom event handler function to process the events.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
order1.EMSX_ORDER_TYPE = 'MKT';
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)
```

```
events =
```

```
  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Route the two existing orders. Specify the order numbers, broker, and hand instruction in the `order` structure. Use a custom event handler function to process the events. You can use the sample event handler function `processEvent` or write your own custom event handler function. For this example, use `processEvent` to process the events.

```
order.EMSX_SEQUENCE{1} = int32(354646);
order.EMSX_SEQUENCE{2} = int32(354777);
order.EMSX_BROKER = 'BB';
order.EMSX_HAND_INSTRUCTION = 'ANY';
groupRouteOrder(c,order,'useDefaultEventHandler',false)
processEvent(c)
```

```
Route = {

     EMSX_SEQUENCE = 354646

     EMSX_ROUTE_ID = 1

     MESSAGE = 'Order Routed'

     }
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Group of Bloomberg EMSX Orders Using Options Structure

Using a Bloomberg EMSX connection, route a group of Bloomberg EMSX orders. Specify an additional option for a timeout value by using a structure.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
order1.EMSX_ORDER_TYPE = 'MKT';
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)

events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Route the two existing orders. Specify the order numbers, broker, and hand instruction in the `order` structure. Specify an additional option for a timeout value of 200 milliseconds by using the `options` structure.

```
order.EMSX_SEQUENCE{1} = int32(354646);
order.EMSX_SEQUENCE{2} = int32(354777);
order.EMSX_BROKER = 'BB';
order.EMSX_HAND_INSTRUCTION = 'ANY';
options.timeOut = 200;
events = groupRouteOrder(c,order,options)

events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    EMSX_ROUTE_ID: 1
    MESSAGE: 'Order Routed'
```

`events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order numbers
- `EMSX_ROUTE_ID` — Bloomberg EMSX route identifier
- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

`order` — Order request

structure

Order request, specified as a structure that contains these fields:

- `EMSX_SEQUENCE` — Order numbers
- `EMSX_BROKER` — Broker
- `EMSX_HAND_INSTRUCTION` — Hand instruction

Convert the order numbers to a 32-bit signed integer by using `int32`.

Data Types: `struct`

`timeout` — Timeout value

`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value

structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `routes` | `close` | `createOrder` | `routeOrder` | `createBasket` | `modifyOrder` | `deleteOrder` | `processEvent`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

groupRouteOrderWithStrat

Route group of Bloomberg EMSX orders with strategies

Syntax

```
events = groupRouteOrderWithStrat(c,route,strat)
events = groupRouteOrderWithStrat(c,route,strat,'timeOut',timeout)

groupRouteOrderWithStrat( ___ ,'useDefaultEventHandler',false)

 ___  = groupRouteOrderWithStrat(c,route,strat,options)
```

Description

`events = groupRouteOrderWithStrat(c,route,strat)` routes multiple Bloomberg EMSX orders with strategies using the Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface, route request `route`, and strategy `strat`. `routeOrderWithStrat` returns the order sequence number, route number, and status message using the default event handler.

`events = groupRouteOrderWithStrat(c,route,strat,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`groupRouteOrderWithStrat( ___ ,'useDefaultEventHandler',false)` routes multiple Bloomberg EMSX orders with strategies using any of the input arguments in the previous syntaxes and a custom event handler. To process the events associated with routing orders, write a custom event handler. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

` ___  = groupRouteOrderWithStrat(c,route,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true`, and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Route Orders Using the Default Event Handler

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies these route request fields:

- Order numbers 335877 and 335878
- Stock symbol IBM
- 100 percent of shares shown on the order to be routed
- Broker BMTB
- Any hand instruction
- Time in force set to DAY
- Market order type

```
route.EMSX_SEQUENCE = {int32(335877);int32(335878)};
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT_PERCENT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Create the order strategies structure `strat` using the strategy SSP. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Route the orders using the Bloomberg EMSX connection `c`, `route`, and `strat`.

```
events = groupRouteOrderWithStrat(c,route,strat)
```

```
events =

     EMSX_SUCCESS_ROUTES: [1x1 struct]
      EMSX_FAILED_ROUTES: [1x1 struct]
                 MESSAGE: '1 of 1 Order(s) Routed'

   where

   events.EMSX_SUCCESS_ROUTES =

        EMSX_SEQUENCE: 335877
        EMSX_ROUTE_ID: 1

   and events.EMSX_FAILED_ROUTES =

        EMSX_SEQUENCE: 335878
           ERROR_CODE: 0
        ERROR_MESSAGE: {'Order 335878  View-only orders can not be routed'}
```

The default event handler processes the events associated with routing the order. `groupRouteOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX success routing structure, which contains the order number and route identifier for the orders that successfully routed
- Bloomberg EMSX failed routing structure, which contains the order number, error code, and error message for the orders that failed to route
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Orders Using a Timeout

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies these route request fields:

- Order numbers `335877` and `335878`
- Stock symbol IBM
- 100 percent of shares shown on the order to be routed
- Broker `BMTB`
- Any hand instruction
- Time in force set to `DAY`
- Market order type

```
route.EMSX_SEQUENCE = {int32(335877);int32(335878)};
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT_PERCENT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Route the orders using the Bloomberg EMSX connection `c`, `route`, and `strat`. Set the timeout value to 200 milliseconds.

```
events = groupRouteOrderWithStrat(c,route,strat,'timeOut',200)
```

```
events =

     EMSX_SUCCESS_ROUTES: [1x1 struct]
      EMSX_FAILED_ROUTES: [1x1 struct]
                 MESSAGE: '1 of 1 Order(s) Routed'

   where

   events.EMSX_SUCCESS_ROUTES =

        EMSX_SEQUENCE: 335877
        EMSX_ROUTE_ID: 1

   and events.EMSX_FAILED_ROUTES =

        EMSX_SEQUENCE: 335878
```

```
    ERROR_CODE: 0
ERROR_MESSAGE: {'Order 335878  View-only orders can not be routed'}
```

The default event handler processes the events associated with routing the order. `groupRouteOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX success routing structure, which contains the order number and route identifier for the orders that successfully routed
- Bloomberg EMSX failed routing structure, which contains the order number, error code, and error message for the orders that failed to route
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Orders Using a Custom Event Handler

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies these route request fields:

- Order numbers `335877` and `335878`
- Stock symbol IBM
- 100 percent of shares shown on the order to be routed
- Broker `BMTB`
- Any hand instruction
- Time in force set to `DAY`
- Market order type

```
route.EMSX_SEQUENCE = {int32(335877);int32(335878)};
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT_PERCENT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Suppose that you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. To run `eventhandler` immediately, start the timer using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Route the orders using the Bloomberg EMSX connection `c`, `route`, and `strat`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with routing an order.

```
groupRouteOrderWithStrat(c,route,strat,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. To stop data updates, stop the timer using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Orders Using an Options Structure

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies these route request fields:

- Order numbers `335877` and `335878`
- Stock symbol IBM
- 100 percent of shares shown on the order to be routed
- Broker `BMTB`
- Any hand instruction
- Time in force set to `DAY`

- Market order type

```
route.EMSX_SEQUENCE = {int32(335877);int32(335878)};
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT_PERCENT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Route the orders using the Bloomberg EMSX connection `c`, `route`, `strat`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = groupRouteOrderWithStrat(c,route,strat,options)
```

```
events =

     EMSX_SUCCESS_ROUTES: [1x1 struct]
      EMSX_FAILED_ROUTES: [1x1 struct]
                 MESSAGE: '1 of 1 Order(s) Routed'

   where

   events.EMSX_SUCCESS_ROUTES =

        EMSX_SEQUENCE: 335877
        EMSX_ROUTE_ID: 1

   and events.EMSX_FAILED_ROUTES =

        EMSX_SEQUENCE: 335878
           ERROR_CODE: 0
        ERROR_MESSAGE: {'Order 335878  View-only orders can not be routed'}
```

The default event handler processes the events associated with routing the order. `groupRouteOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX success routing structure, which contains the order number and route identifier for the orders that successfully routed
- Bloomberg EMSX failed routing structure, which contains the order number, error code, and error message for the orders that failed to route
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

route — Route request

structure

Route request, specified as a structure containing these fields.

Convert the numbers to 32-bit signed integers using `int32`. `EMSX_SEQUENCE` must denote an existing order sequence number.

Field	Description
`EMSX_SEQUENCE`	Bloomberg EMSX order sequence number
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX number of shares
`EMSX_BROKER`	Bloomberg EMSX broker name
`EMSX_HAND_INSTRUCTION`	Bloomberg EMSX hand instruction
`EMSX_TIF`	Bloomberg EMSX time in force
`EMSX_ORDER_TYPE`	Bloomberg EMSX order type

Example:
```
route.EMSX_SEQUENCE = int32(728918);
route.EMSX_TICKER = 'XYZ';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Data Types: `struct`

strat — Order strategies

structure

Order strategies, specified as a structure that contains the fields: `EMSX_STRATEGY_NAME`, `EMSX_STRATEGY_FIELD_INDICATORS`, and `EMSX_STRATEGY_FIELDS`. The structure field values must align with the strategy fields specified by `EMSX_STRATEGY_NAME`. For details about strategy fields and ordering, see `getBrokerInfo`.

Convert `EMSX_STRATEGY_FIELD_INDICATORS` to a 32-bit signed integer using `int32`. Set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `0` for each field to use the field data setting in `EMSX_FIELD_DATA`. Or, set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `1` to ignore the data in `EMSX_FIELD_DATA`.

Example:
```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Data Types: `struct`

`timeout` — Timeout value
`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `routes` | `close` | `createOrder` | `routeOrder` | `routeOrderWithStrat` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `modifyOrder` | `deleteOrder` | `deleteRoute` | `getBrokerInfo` | `timer` | `start` | `stop` | `delete`

Topics
“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites
EMSX API Programmers Guide

manualFill

Fill Bloomberg EMSX orders manually

Syntax

```
events = manualFill(c,order)
events = manualFill(c,order,'timeOut',timeout)
manualFill( ___ ,'useDefaultEventHandler',false)
 ___  = manualFill(c,order,options)
```

Description

`events = manualFill(c,order)` manually fills a Bloomberg EMSX order using the Bloomberg EMSX connection with the Bloomberg EMSX C++ interface and order request. `manualFill` returns the order sequence number and status message using the default event handler.

`events = manualFill(c,order,'timeOut',timeout)` specifies a timeout value for the execution of the default event handler.

`manualFill( ___ ,'useDefaultEventHandler',false)` manually fills a Bloomberg EMSX order using any of the previous input argument combinations and a custom event handler function. Write a custom event handler to process the events associated with manually filling an order. This syntax does not have an output argument because the custom event handler processes the contents of the event queue.

`___  = manualFill(c,order,options)` uses the `options` structure to customize the output, which is useful for configuring and saving your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the `useDefaultEventHandler` field is set to `true`, and omit this output argument when the `useDefaultEventHandler` field is set to `false`.

Examples

Manually Fill Bloomberg EMSX Order Using Default Event Handler

Using a Bloomberg EMSX connection, manually fill a Bloomberg EMSX order.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
```

```
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order`.

```
events = createOrder(c,order)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Manually fill the Bloomberg order. Specify the `manualorder` structure with the order number in the `events` structure.

```
manualorder.EMSX_SEQUENCE = int32(events.EMSX_SEQUENCE);
events = manualFill(c,manualorder)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order Filled'
```

The default event handler processes the events associated with manually filling the order. `events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Manually Fill Bloomberg EMSX Order Using Timeout Value

Using a Bloomberg EMSX connection, manually fill a Bloomberg EMSX order. Specify a timeout value.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force

set to DAY and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order`.

```
events = createOrder(c,order)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Manually fill the Bloomberg order. Specify the `manualorder` structure with the order number in the `events` structure. Specify the timeout value of 200 milliseconds by using the `'timeOut'` flag.

```
manualorder.EMSX_SEQUENCE = int32(events.EMSX_SEQUENCE);
events = manualFill(c,manualorder,'timeOut',200)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order Filled'
```

`events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Manually Fill Bloomberg EMSX Order Using Custom Event Handler Function

Using a Bloomberg EMSX connection, manually fill a Bloomberg EMSX order. Specify using a custom event handler function to process the events.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order`.

```
events = createOrder(c,order)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Manually fill the Bloomberg order. Specify the `manualorder` structure with the order number in the `events` structure. Use a custom event handler function to process the events. You can use the sample event handler function `processEvent` or write your own custom event handler function. For this example, use `processEvent` to process the events.

```
manualorder.EMSX_SEQUENCE = int32(events.EMSX_SEQUENCE);
manualFill(c,manualorder,'useDefaultEventHandler',false)
processEvent(c)
```

```
ManualFill = {

    EMSX_SEQUENCE = 354646

    MESSAGE = 'Order Filled'

}
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Manually Fill Bloomberg EMSX Order Using Options Structure

Using a Bloomberg EMSX connection, manually fill a Bloomberg EMSX order. Specify an additional option for a timeout value by using a structure.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Create the order request structure `order` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order`.

```
events = createOrder(c,order)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Manually fill the Bloomberg order. Specify the `manualorder` structure with the order number in the `events` structure. Then, specify an additional option for a timeout value of 200 milliseconds by using the `options` structure.

```
manualorder.EMSX_SEQUENCE = int32(events.EMSX_SEQUENCE);
options.timeOut = 200;
events = manualFill(c,manualorder,options)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order Filled'
```

`events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number

- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

`order` — Order request

structure

Order request, specified as a structure that contains the `EMSX_SEQUENCE` field. This field contains the order numbers. Convert the order numbers to a 32-bit signed integer by using `int32`.

Example: `int32(123456)`

Data Types: `struct`

`timeout` — Timeout value

`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value

structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `routes` | `close` | `createOrder` | `routeOrder` | `groupRouteOrder` | `createBasket` | `modifyOrder` | `deleteOrder` | `processEvent`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

modifyOrder

Modify Bloomberg EMSX order

Syntax

```
events = modifyOrder(c,modorder)
events = modifyOrder(c,modorder,'timeOut',timeout)

modifyOrder( ___ ,'useDefaultEventHandler',false)

 ___  = modifyOrder(c,modorder,options)
```

Description

`events = modifyOrder(c,modorder)` modifies a Bloomberg EMSX order using the Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface and modify order request structure `modorder`. `modifyOrder` returns a status message using the default event handler.

`events = modifyOrder(c,modorder,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`modifyOrder( ___ ,'useDefaultEventHandler',false)` modifies a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with modifying orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___  = modifyOrder(c,modorder,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Modify an Order Using the Default Event Handler

To modify a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number `728905` for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Modify the order using the Bloomberg EMSX connection `c` and `modorder`.

```
events = modifyOrder(c,modorder)
```

```
events =

    EMSX_SEQUENCE: 728905
          MESSAGE: 'Order Modified'
```

The default event handler processes the events associated with modifying the order. `modifyOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify an Order Using a Timeout

To modify a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number `728905` for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Modify the order using the Bloomberg EMSX connection `c` and `modorder`. Set the timeout value to 200 milliseconds.

```
events = modifyOrder(c,modorder,'timeOut',200)
```

```
events =

    EMSX_SEQUENCE: 728905
          MESSAGE: 'Order Modified'
```

The default event handler processes the events associated with modifying the order. `modifyOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify an Order Using a Custom Event Handler

To modify a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number `728905` for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Modify the order using the Bloomberg EMSX connection `c` and `modorder`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with modifying an order.

```
modifyOrder(c,modorder,'useDefaultEventHandler',false)
```

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(subs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify an Order Using an Options Structure

To modify a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number `728905` for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Modify the order using the Bloomberg EMSX connection `c`, `modorder`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = modifyOrder(c,modorder,options)
```

```
events =

    EMSX_SEQUENCE: 728905
          MESSAGE: 'Order Modified'
```

The default event handler processes the events associated with modifying the order. `modifyOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

`modorder` — Modify order request

structure

Modify order request, specified as a structure that contains these fields.

Use `getAllFieldMetaData` to view all available fields for `modorder`. Convert the numbers to 32-bit signed integers using `int32`.

Field	Description
`EMSX_SEQUENCE`	Bloomberg EMSX order sequence number
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX number of shares

Example:
```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'XYZ';
modorder.EMSX_AMOUNT = int32(100);
```

Data Types: `struct`

`timeout` — Timeout value
`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `routes` | `close` | `createOrder` | `routeOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `deleteOrder` | `deleteRoute` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

modifyRoute

Modify Bloomberg EMSX route

Syntax

```
events = modifyRoute(c,modroute)
events = modifyRoute(c,modroute,'timeOut',timeout)

modifyRoute( ___ ,'useDefaultEventHandler',false)

 ___  = modifyRoute(c,modroute,options)
```

Description

`events = modifyRoute(c,modroute)` modifies a Bloomberg EMSX route using the Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface and route request `modroute`. `modifyRoute` returns a status message using the default event handler.

`events = modifyRoute(c,modroute,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`modifyRoute( ___ ,'useDefaultEventHandler',false)` modifies a Bloomberg EMSX route using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with modifying routes. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___  = modifyRoute(c,modroute,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Modify a Route Using the Default Event Handler

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code instructs Bloomberg EMSX to route 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Modify the route using the Bloomberg EMSX connection `c` and `modroute`.

```
events = modifyRoute(c,modroute)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route Using a Timeout

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Modify the route using the Bloomberg EMSX connection `c` and `modroute`. Set the timeout value to 200 milliseconds.

```
events = modifyRoute(c,modroute,'timeOut',200)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route Using a Custom Event Handler

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Modify the route using the Bloomberg EMSX connection `c` and `modroute`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with modifying a route.

```
modifyRoute(c,modroute,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route Using an Options Structure

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = modifyRoute(c,modroute,options)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

`modroute` — Modify route request
structure

Modify route request, specified as a structure with these fields.

Use `getAllFieldMetaData` to view all available fields for `modroute`. Convert the numbers to 32-bit signed integers using `int32`.

Field	Description
`EMSX_SEQUENCE`	Bloomberg EMSX order sequence number
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX number of shares
`EMSX_ROUTE_ID`	Bloomberg EMSX route identifier

Example:
```
modroute.EMSX_SEQUENCE = int32(731128);
modroute.EMSX_TICKER = 'XYZ';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Data Types: `struct`

`timeout` — Timeout value
`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`orders` | `routes` | `close` | `createOrder` | `createOrderAndRoute` | `modifyRouteWithStrat` | `deleteOrder` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

modifyRouteWithStrat

Modify Bloomberg EMSX route with strategies

Syntax

```
events = modifyRouteWithStrat(c,modroute,strat)
events = modifyRouteWithStrat(c,modroute,strat,'timeOut',timeout)

modifyRouteWithStrat( ___ ,'useDefaultEventHandler',false)

 ___  = modifyRouteWithStrat(c,modroute,strat,options)
```

Description

`events = modifyRouteWithStrat(c,modroute,strat)` modifies a Bloomberg EMSX route with strategies using the Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface, route request `modroute`, and order strategy `strat`. `modifyRouteWithStrat` returns the order sequence number, route identifier, and status message using the default event handler.

`events = modifyRouteWithStrat(c,modroute,strat,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`modifyRouteWithStrat( ___ ,'useDefaultEventHandler',false)` modifies a Bloomberg EMSX route with strategies using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with modifying routes. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

` ___  = modifyRouteWithStrat(c,modroute,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Modify a Route with Strategies Using the Default Event Handler

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and `strat`.

```
events = modifyRouteWithStrat(c,modroute,strat)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route with Strategies Using a Timeout

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.

- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and `strat`. Set the timeout value to 200 milliseconds.

```
events = modifyRouteWithStrat(c,modroute,strat,'timeOut',200)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route with Strategies Using a Custom Event Handler

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and `strat`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with modifying a route.

```
modifyRouteWithStrat(c,modroute,strat,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route with Strategies Using an Options Structure

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Modify the route using the Bloomberg EMSX connection `c`, `modroute`, `strat`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = modifyRouteWithStrat(c,modroute,strat,options)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

`modroute` — Modify route request

structure

Modify route request, specified as a structure with these fields.

Use `getAllFieldMetaData` to view all available fields for `modroute`. Convert the numbers to 32-bit signed integers using `int32`.

Field	Description
`EMSX_SEQUENCE`	Bloomberg EMSX order sequence number
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX number of shares
`EMSX_ROUTE_ID`	Bloomberg EMSX route identifier

Example: `modroute.EMSX_SEQUENCE = int32(731128);`

```
modroute.EMSX_TICKER = 'XYZ';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Data Types: `struct`

`strat` — Order strategies

structure

Order strategies, specified as a structure that contains the fields: `EMSX_STRATEGY_NAME`, `EMSX_STRATEGY_FIELD_INDICATORS`, and `EMSX_STRATEGY_FIELDS`. The structure field values must align with the strategy fields specified by `EMSX_STRATEGY_NAME`. For details about strategy fields and ordering, see `getBrokerInfo`.

Convert `EMSX_STRATEGY_FIELD_INDICATORS` to a 32-bit signed integer using `int32`. Set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `0` for each field to use the field data setting in `EMSX_FIELD_DATA`. Or, set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `1` to ignore the data in `EMSX_FIELD_DATA`.

Example:
```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Data Types: `struct`

`timeout` — Timeout value
`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`orders` | `routes` | `close` | `createOrder` | `routeOrder` | `createOrderAndRouteWithStrat` | `modifyRoute` | `deleteOrder` | `getBrokerInfo` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

orders

Obtain Bloomberg EMSX order subscription

Syntax

```
events = orders(c,fields)

events = orders(c,fields,Name,Value)
events = orders(c,fields,options)
```

Description

`events = orders(c,fields)` subscribes to Bloomberg EMSX fields `fields` using the Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface. `orders` returns existing event data `events` from the event queue.

`events = orders(c,fields,Name,Value)` uses additional options specified by one or more `Name`,`Value` pair arguments to specify a custom event handler or timeout value for the event handler.

`events = orders(c,fields,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The `options` structure fields and values correspond to names and values of name-value pair arguments, respectively.

Examples

Subscribe to Order Events Using Default Event Handler

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c` and Bloomberg EMSX field list `fields`.

```
fields = {'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'};

events = orders(c,fields)

events =

                    MSG_TYPE: {'E'}
                MSG_SUB_TYPE: {'O'}
                EVENT_STATUS: 4
                ...
```

`events` contains fields for the events currently in the event queue.

Unsubscribe from order events using the Bloomberg EMSX subscriptions.

```
c.Session.stopSubscriptions
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Subscribe to Order Events Using Custom Event Handler

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Write a custom event handler function named `eventhandler`. Run the custom event handler using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1, ...
    'ExecutionMode','fixedRate');
start(t)
```

`t` is the timer object.

Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c` and Bloomberg EMSX field list `fields`. Use the custom event handler by setting the name-value pair argument `'useDefaultEventHandler'` to `false`.

```
fields = {'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'};

events = orders(c,fields,'useDefaultEventHandler',false)
```

```
events =

     []
```

`events` contains an empty double. The custom event handler processes the event queue.

Unsubscribe from order events using the Bloomberg EMSX subscriptions. Stop the timer to stop data updates using `stop`.

```
c.Session.stopSubscriptions
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Subscribe to Order Events Using Timeout

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c` and Bloomberg EMSX field list `fields`. Specify the name-value pair argument `timeOut` and set it to 200 milliseconds.

```
fields = {'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'};

events = orders(c,fields,'timeOut',200)
```

```
events =

                    MSG_TYPE: {'E'}
                MSG_SUB_TYPE: {'O'}
                EVENT_STATUS: 4
                ...
```

`events` contains fields for the events currently in the event queue.

Unsubscribe from order events using the Bloomberg EMSX subscriptions.

```
c.Session.stopSubsriptions
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Subscribe to Order Events Using the Options Structure

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c`, Bloomberg EMSX field list `fields`, and options structure `options`.

```
options.timeOut = 200;
options.useDefaultEventHandler = true;

fields = {'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'};

events = orders(c,fields,options)
```

```
events =

                    MSG_TYPE: {'E'}
                MSG_SUB_TYPE: {'O'}
                EVENT_STATUS: 4
                ...
```

`events` contains fields for the events currently in the event queue.

Unsubscribe from order events using the Bloomberg EMSX subscriptions.

```
c.Session.stopSubscriptions
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

fields — Bloomberg EMSX field information

cell array

Bloomberg EMSX field information, specified using a cell array. Use `getAllFieldMetaData` to view available field information for the Bloomberg EMSX service.

Example: `'EMSX_TICKER'`
`'EMSX_AMOUNT'`
`'EMSX_ORDER_TYPE'`

Data Types: `cell`

options — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure instead of name-value pair arguments to reuse the optional name-value pair arguments to specify a custom event handler or timeout value for the event handler.

The `options` structure field and values correspond to names and values of the name-value pair arguments, respectively.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example:
```
options.useDefaultEventHandler = false;
options.timeOut = 500;
```

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'useDefaultEventHandler',false`

useDefaultEventHandler — Flag for event handler preference

`true` (default) | `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, specified as the comma-separated pair consisting of `'useDefaultEventHandler'` and the logical values `true` or `false`.

To specify the default event handler, set this flag to `true`.

Otherwise, set this flag to `false` to specify a custom event handler.

Data Types: `logical`

`timeOut` — Timeout value for event handler
`500` milliseconds (default) | nonnegative integer

Timeout value for event handler for the Bloomberg EMSX service, specified as the comma-separated pair consisting of `'timeOut'` and a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types: `double`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

When the name-value pair argument `'useDefaultEventHandler'` or the same field for the structure `options` is set to `false`, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `routes` | `close` | `createOrder` | `routeOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `modifyOrder` | `deleteOrder` | `deleteRoute` | `getAllFieldMetaData` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

processEvent

Sample Bloomberg EMSX event handler

Syntax

```
processEvent(c)
```

Description

`processEvent(c)` displays and flushes the event queue associated with the Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface. `processEvent` is a sample event handler function. You can build a custom event handler function to process Bloomberg EMSX events.

Examples

Continually Process the Bloomberg EMSX Event Queue

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Use `timer` to continually process the Bloomberg EMSX event queue.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
```

`t` is the MATLAB timer object. For details, see `timer`.

Close the Bloomberg EMSX connection.

```
close(c)
```

Process the Bloomberg EMSX Event Queue Once

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Use the default event handler function `processEvent` to process the Bloomberg EMSX event queue once.

```
processEvent(c)
```

```
SessionConnectionUp = {

    server = "localhost/127.0.0.1:8194"

}

SessionStarted = {
```

```
}

ServiceOpened = {

    serviceName = "//blp/emapisvc_beta"

}
```

`processEvent` clears the Bloomberg EMSX event queue.

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `routes` | `close` | `createOrder` | `routeOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `modifyOrder` | `deleteOrder` | `deleteRoute` | `timer`

Topics

“Create Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-2
“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

routeOrder

Route Bloomberg EMSX order

Syntax

```
events = routeOrder(c,route)
events = routeOrder(c,route,'timeOut',timeout)

routeOrder( ___ ,'useDefaultEventHandler',false)

 ___  = routeOrder(c,route,options)
```

Description

`events = routeOrder(c,route)` routes a Bloomberg EMSX order using the Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface and route request `route`. `routeOrder` returns a status message using the default event handler.

`events = routeOrder(c,route,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`routeOrder( ___ ,'useDefaultEventHandler',false)` routes a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___  = routeOrder(c,route,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Route an Order Using the Default Event Handler

To route a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
```

```
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Route the order using the Bloomberg EMSX connection `c` and `route`.

```
events = routeOrder(c,route)
```

```
events =

     EMSX_SEQUENCE: 335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using a Timeout

To route a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Route the order using the Bloomberg EMSX connection `c` and `route`. Set the timeout value to 200 milliseconds.

```
events = routeOrder(c,route,'timeOut',200)
```

```
events =
```

```
    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using a Custom Event Handler

To route a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Route the order using the Bloomberg EMSX connection `c` and `route`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with routing an order.

```
routeOrder(c,route,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using an Options Structure

To route a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Route the order using the Bloomberg EMSX connection `c`, `route`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = routeOrder(c,route,options)
```

```
events =

     EMSX_SEQUENCE: 335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

`route` — Route request

structure

Route request, specified as a structure containing these fields.

Convert the numbers to 32-bit signed integers using `int32`. `EMSX_SEQUENCE` must denote an existing order sequence number.

Field	Description
`EMSX_SEQUENCE`	Bloomberg EMSX order sequence number
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX number of shares
`EMSX_BROKER`	Bloomberg EMSX broker name
`EMSX_HAND_INSTRUCTION`	Bloomberg EMSX hand instruction

Example:
```
route.EMSX_SEQUENCE = int32(728918);
route.EMSX_TICKER = 'XYZ';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Data Types: `struct`

`timeout` — Timeout value

`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value

structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `routes` | `close` | `createOrder` | `routeOrderWithStrat` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `modifyOrder` | `deleteOrder` | `deleteRoute` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

routeOrderWithStrat

Route Bloomberg EMSX order with strategies

Syntax

```
events = routeOrderWithStrat(c,route,strat)
events = routeOrderWithStrat(c,route,strat,'timeOut',timeout)

routeOrderWithStrat( ___ ,'useDefaultEventHandler',false)

 ___  = routeOrderWithStrat(c,route,strat,options)
```

Description

`events = routeOrderWithStrat(c,route,strat)` routes a Bloomberg EMSX order with strategies using the Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface, route request `route`, and strategy `strat`. `routeOrderWithStrat` returns the order sequence number, route number, and status message using the default event handler.

`events = routeOrderWithStrat(c,route,strat,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`routeOrderWithStrat( ___ ,'useDefaultEventHandler',false)` routes a Bloomberg EMSX order with strategies using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___  = routeOrderWithStrat(c,route,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Route an Order Using the Default Event Handler

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Route the order using the Bloomberg EMSX connection `c`, `route`, and `strat`.

```
events = routeOrderWithStrat(c,route,strat)
```

```
events =

     EMSX_SEQUENCE: 335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using a Timeout

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
```

```
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Route the order using the Bloomberg EMSX connection `c`, `route`, and `strat`. Set the timeout value to 200 milliseconds.

```
events = routeOrderWithStrat(c,route,strat,'timeOut',200)
```

```
events =

     EMSX_SEQUENCE: 335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using a Custom Event Handler

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Route the order using the Bloomberg EMSX connection `c`, `route`, and `strat`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with routing an order.

```
routeOrderWithStrat(c,route,strat,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using an Options Structure

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Route the order using the Bloomberg EMSX connection `c`, `route`, `strat`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = routeOrderWithStrat(c,route,strat,options)
```

```
events =

     EMSX_SEQUENCE: 335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

`route` — Route request

structure

Route request, specified as a structure containing these fields.

Convert the numbers to 32-bit signed integers using `int32`. `EMSX_SEQUENCE` must denote an existing order sequence number.

Field	Description
EMSX_SEQUENCE	Bloomberg EMSX order sequence number
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX number of shares
EMSX_BROKER	Bloomberg EMSX broker name
EMSX_HAND_INSTRUCTION	Bloomberg EMSX hand instruction

Example:
```
route.EMSX_SEQUENCE = int32(728918);
route.EMSX_TICKER = 'XYZ';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Data Types: `struct`

`strat` — Order strategies
structure

Order strategies, specified as a structure that contains the fields: `EMSX_STRATEGY_NAME`, `EMSX_STRATEGY_FIELD_INDICATORS`, and `EMSX_STRATEGY_FIELDS`. The structure field values must align with the strategy fields specified by `EMSX_STRATEGY_NAME`. For details about strategy fields and ordering, see `getBrokerInfo`.

Convert `EMSX_STRATEGY_FIELD_INDICATORS` to a 32-bit signed integer using `int32`. Set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `0` for each field to use the field data setting in `EMSX_FIELD_DATA`. Or, set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `1` to ignore the data in `EMSX_FIELD_DATA`.

Example:
```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Data Types: `struct`

`timeout` — Timeout value
`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `routes` | `close` | `createOrder` | `routeOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `modifyOrder` | `deleteOrder` | `deleteRoute` | `getBrokerInfo` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

routes

Obtain Bloomberg EMSX route subscription

Syntax

```
events = routes(c,fields)

events = routes(c,fields,Name,Value)
events = routes(c,fields,options)
```

Description

`events = routes(c,fields)` subscribes to Bloomberg EMSX fields `fields` using the Bloomberg EMSX connection `c` with the Bloomberg EMSX C++ interface. `routes` returns existing event data `events` from the event queue.

`events = routes(c,fields,Name,Value)` uses additional options specified by one or more `Name`,`Value` pair arguments to specify a custom event handler or timeout value for the event handler.

`events = routes(c,fields,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The `options` structure fields and values correspond to names and values of name-value pair arguments, respectively.

Examples

Set Up Route Subscription Using Default Event Handler

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};

events = routes(c,fields)
```

```
events =

                    MSG_TYPE: {5x1 cell}
                MSG_SUB_TYPE: {5x1 cell}
                EVENT_STATUS: [5x1 int32]
                ...
```

`events` contains fields for the events currently in the event queue.

Unsubscribe from route events using the Bloomberg EMSX subscriptions.

```
c.Session.stopSubscriptions
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Set Up Route Subscription Using Custom Event Handler

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Write a custom event handler function named `eventhandler`. Run the custom event handler using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1, ...
          'ExecutionMode','fixedRate');
start(t)
```

`t` is the timer object.

Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`. Use the custom event handler by setting the name-value pair argument `'useDefaultEventHandler'` to `false`.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};

events = routes(c,fields,'useDefaultEventHandler',false)

events =

     []
```

`events` is an empty double. The custom event handler processes the event queue.

Unsubscribe from route events using the Bloomberg EMSX subscriptions. Stop the timer to stop data updates using `stop`.

```
c.Session.stopSubscriptions
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Set Up Route Subscription Using Timeout

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`. Specify the name-value pair argument `timeOut` and set it to 200 milliseconds.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};

events = routes(c,fields,'timeOut',200)
```

```
events =

                    MSG_TYPE: {5x1 cell}
                MSG_SUB_TYPE: {5x1 cell}
                EVENT_STATUS: [5x1 int32]
                ...
```

`events` contains fields for the events currently in the event queue.

Unsubscribe from route events using the Bloomberg EMSX subscriptions.

```
c.Session.stopSubscriptions
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Set Up Route Subscription Using Options Structure

Create the Bloomberg EMSX connection `c` using the Bloomberg EMSX C++ interface.

```
c = bloombergEMSX('//blp/emapisvc_beta');
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c` and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;
fields = {'EMSX_BROKER','EMSX_WORKING'};

events = routes(c,fields,options)
```

```
events =

                    MSG_TYPE: {5x1 cell}
                MSG_SUB_TYPE: {5x1 cell}
                EVENT_STATUS: [5x1 int32]
                ...
```

`events` contains fields for the events currently in the event queue.

Unsubscribe from route events using the Bloomberg EMSX subscriptions.

```
c.Session.stopSubscriptions
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

`bloombergEMSX` object

Bloomberg EMSX service connection, specified as a `bloombergEMSX` object.

fields — Bloomberg EMSX field information

cell array

Bloomberg EMSX field information, specified using a cell array. Use `getAllFieldMetaData` to view available field information for the Bloomberg EMSX service.

Example: `'EMSX_TICKER'`
`'EMSX_AMOUNT'`
`'EMSX_ORDER_TYPE'`

Data Types: `cell`

options — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure instead of name-value pair arguments to reuse the optional name-value pair arguments to specify a custom event handler or timeout value for the event handler.

The `options` structure field and values correspond to names and values of the name-value pair arguments, respectively.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example:
```
options.useDefaultEventHandler = false;
options.timeOut = 500;
```

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'useDefaultEventHandler',false`

useDefaultEventHandler — Flag for event handler preference

`true` (default) | `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, specified as the comma-separated pair consisting of `'useDefaultEventHandler'` and the logical values `true` or `false`.

To specify the default event handler, set this flag to `true`.

Otherwise, set this flag to `false` to specify a custom event handler.

Data Types: `logical`

`timeOut` — Timeout value for event handler
`500` milliseconds (default) | nonnegative integer

Timeout value for event handler for the Bloomberg EMSX service, specified as the comma-separated pair consisting of `'timeOut'` and a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types: `double`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

When the name-value pair argument `'useDefaultEventHandler'` or the same field for the structure `options` is set to `false`, `events` is an empty double.

Tips

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using this code.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1, ...
          'ExecutionMode','fixedRate')
```

`t` is the MATLAB timer object. For details, see `timer`.

Version History

Introduced in R2021a

See Also

`bloombergEMSX` | `orders` | `close` | `createOrder` | `routeOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `modifyOrder` | `modifyRoute` | `deleteOrder` | `deleteRoute` | `getAllFieldMetaData` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order Using Bloomberg EMSX C++ Interface” on page 5-4
“Create and Manage Bloomberg EMSX Route Using Bloomberg EMSX C++ Interface” on page 5-7
“Manage Bloomberg EMSX Order and Route Using Bloomberg EMSX C++ Interface” on page 5-11
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites
EMSX API Programmers Guide

emsx

Create Bloomberg EMSX connection

Description

The `emsx` function creates an `emsx` object, which represents a Bloomberg EMSX connection. After you create an `emsx` object, you can use the object functions to create and route orders, and manage orders and routes. For details about Bloomberg EMSX, see the *EMSX API Programmers Guide*.

Creation

Syntax

```
c = emsx(servicename)

c = emsx(servicename,authid,serverip)
c = emsx(servicename,authid,serverip,portnumber)
c = emsx(servicename,authid,serverip,portnumber,terminalip)
```

Description

Local Connection

`c = emsx(servicename)` creates a connection to the local Bloomberg EMSX communications server using the service `servicename`.

Remote Connection

`c = emsx(servicename,authid,serverip)` creates a connection to a remote EMSX server using the specified service name, authentication identifier, and server IP address.

`c = emsx(servicename,authid,serverip,portnumber)` also specifies the port number for the remote connection.

`c = emsx(servicename,authid,serverip,portnumber,terminalip)` also specifies the IP address of the machine you use to access the Bloomberg Terminal for the remote connection.

Input Arguments

`servicename` — Bloomberg EMSX service name

`'//blp/emapisvc_beta'` | `'//blp/emapisvc'`

Bloomberg EMSX service name, specified as one of these connection types.

Connection Type	Bloomberg EMSX Service Name
Test	`'//blp/emapisvc_beta'`
Production	`'//blp/emapisvc'`

`authid` — Bloomberg EMSX authentication identifier
character vector | string scalar

Bloomberg EMSX authentication identifier, specified as a character vector or string scalar.

For Bloomberg EMSX Desktop, specify an empty character vector or string scalar because this input argument is not required. For Bloomberg EMSX Server, this input argument is required.

`serverip` — Bloomberg EMSX Server IP address
character vector | string scalar

Bloomberg EMSX Server IP address, specified as a character vector or string scalar. This address is the IP address of the machine running the Bloomberg EMSX Server process.

For Bloomberg EMSX Desktop, specify an empty character vector or string scalar because this input argument is not required. For Bloomberg EMSX Server, this input argument is required.

Example: `'111.222.333.44'`

`portnumber` — Port number
`8194` (default) | numeric scalar

Port number of the machine running the EMSX Server process, specified as a numeric scalar.

For Bloomberg EMSX Desktop, specify an empty array because this input argument is not required. For Bloomberg EMSX Server, this input argument is required.

`terminalip` — Bloomberg Terminal IP address
`"localhost"` (default) | character vector | string scalar

Bloomberg Terminal IP address, specified as a character vector or string scalar. This address is the IP address of the machine you use to access the Bloomberg Terminal.

Example: `'111.222.333.44'`

Properties

`Session` — Bloomberg EMSX session
session object

This property is read-only.

Bloomberg EMSX session, specified as a Bloomberg EMSX session object.

Example: `[1x1 com.bloomberglp.blpapi.Session]`

`Service` — Bloomberg EMSX service
service object

This property is read-only.

Bloomberg EMSX service, specified as a Bloomberg EMSX service object.

The `emsx` function sets this property using the `servicename` input argument.

Example: `[1x1 com.bloomberglp.blpapi.impl.aQ]`

`Ipaddress` — IP address
`'localhost'` (default) | character vector

This property is read-only.

IP address of the machine running Bloomberg EMSX, specified as a character vector.

Data Types: `char`

`Port` — Port number
numeric scalar

This property is read-only.

Port number of the machine running Bloomberg EMSX, specified as a numeric scalar.

Example: `8194`

Data Types: `double`

`User` — User
Bloomberg API Java object

This property is read-only.

User, specified as a Bloomberg API Java object for Bloomberg EMSX Server. For Bloomberg EMSX Desktop, this property is empty.

Example: `[1×1 com.bloomberglp.blpapi.impl.by]`

Object Functions

Create Bloomberg EMSX Connection

close	Close Bloomberg EMSX connection
orders	Obtain Bloomberg EMSX order subscription
routes	Obtain Bloomberg EMSX route subscription

Create Bloomberg EMSX Orders and Routes

createOrder	Create Bloomberg EMSX order
routeOrder	Route Bloomberg EMSX order
routeOrderWithStrat	Route Bloomberg EMSX order with strategies
groupRouteOrder	Route group of Bloomberg EMSX orders
groupRouteOrderWithStrat	Route group of Bloomberg EMSX orders with strategies
createOrderAndRoute	Create and route Bloomberg EMSX order
createOrderAndRouteWithStrat	Create and route Bloomberg EMSX order with strategies
createBasket	Create basket of Bloomberg EMSX orders

Manage Bloomberg EMSX Orders and Routes

manualFill	Fill Bloomberg EMSX orders manually
modifyOrder	Modify Bloomberg EMSX order
modifyRoute	Modify Bloomberg EMSX route
modifyRouteWithStrat	Modify Bloomberg EMSX route with strategies
deleteOrder	Delete Bloomberg EMSX order

deleteRoute	Delete Bloomberg EMSX active shares
processEvent	Sample Bloomberg EMSX event handler

Retrieve Bloomberg EMSX Information

emsxOrderBlotter	Bloomberg EMSX example order blotter
getBrokerInfo	Obtain Bloomberg EMSX broker and strategy information
getAllFieldMetaData	Obtain Bloomberg EMSX field information

Examples

Connect to Bloomberg EMSX Test Service

First, create a Bloomberg EMSX test service connection. Then, obtain broker information.

Create a connection `c` to the Bloomberg EMSX test service. You can place test calls using this service.

```
c = emsx('//blp/emapisvc_beta')
```

```
c =

  emsx with properties:

      Session: [1x1 com.bloomberglp.blpapi.Session]
      Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
    Ipaddress: 'localhost'
         Port: 8194
         User: []
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following properties:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service

Define the broker and strategy information structure `brokerstrat`. Obtain broker information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

The `EMSX_BROKERS` field lists the Bloomberg EMSX brokers.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
```

```
r = getBrokerInfo(c,brokerstrat)
```

```
 r =

        EMSX_BROKERS: {2x1 cell}
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Connect to Bloomberg EMSX Production Service

First, create a Bloomberg EMSX production service connection. Then, obtain broker information.

Create a connection `c` to the Bloomberg EMSX production service. You can place live calls using this service.

```
c = emsx('//blp/emapisvc')
```

```
c =

  emsx with properties:

      Session: [1x1 com.bloomberglp.blpapi.Session]
      Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
    Ipaddress: 'localhost'
         Port: 8194
         User: []
```

MATLAB returns `c` as the connection to the Bloomberg EMSX production service with the following properties:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX production service
- Port number of the machine running the Bloomberg EMSX production service

Define the broker and strategy information structure `brokerstrat`. Obtain broker information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

The `EMSX_BROKERS` field lists the Bloomberg EMSX brokers.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
```

```
r = getBrokerInfo(c,brokerstrat)
```

```
 r =

        EMSX_BROKERS: {2x1 cell}
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Connect to Bloomberg EMSX Remote Server

Obtain broker information using a Bloomberg EMSX connection to a remote server.

Create a connection `c` to the Bloomberg EMSX remote server. Specify the service name, authentication identifier, and server IP address.

```
servicename = '//blp/emapisvc_beta';
authid = 'abcdef123';
serverip = '111.222.333.44';
c = emsx(servicename,authid,serverip)
```

```
c =

  emsx with properties:

      Session: [1x1 com.bloomberglp.blpapi.Session]
      Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
    Ipaddress: '111.222.333.44'
         Port: 8194
         User: [1×1 com.bloomberglp.blpapi.impl.by]
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following properties:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service
- Bloomberg API Java object

Define the broker and strategy information structure `brokerstrat`. Obtain broker information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

The `EMSX_BROKERS` field lists the Bloomberg EMSX brokers.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';

r = getBrokerInfo(c,brokerstrat)

 r =

        EMSX_BROKERS: {2x1 cell}
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Connect to Bloomberg EMSX Remote Server with Port Number

Obtain broker information using a Bloomberg EMSX connection to a remote server with a port number.

Create a connection `c` to the Bloomberg EMSX remote server. Specify the service name, authentication identifier, server IP address, and port number.

```
servicename = '//blp/emapisvc_beta';
authid = 'abcdef123';
serverip = '111.222.333.44';
portnumber = 8194;
c = emsx(servicename,authid,serverip,portnumber)

c =

  emsx with properties:
```

```
      Session: [1x1 com.bloomberglp.blpapi.Session]
      Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
    Ipaddress: '111.222.333.44'
         Port: 8194
         User: [1×1 com.bloomberglp.blpapi.impl.by]
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following properties:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service
- Bloomberg API Java object

Define the broker and strategy information structure `brokerstrat`. Obtain broker information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

The `EMSX_BROKERS` field lists the Bloomberg EMSX brokers.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
```

```
r = getBrokerInfo(c,brokerstrat)
```

```
 r =

        EMSX_BROKERS: {2x1 cell}
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Connect to Bloomberg EMSX Remote Server with Terminal IP Address

Obtain broker information using a Bloomberg EMSX connection to a remote server with a port number and Bloomberg Terminal IP address.

Create a connection `c` to the Bloomberg EMSX remote server. Specify the service name, authentication identifier, server IP address, and port number. Also, specify the IP address of the machine you use to access the Bloomberg Terminal.

```
servicename = '//blp/emapisvc_beta';
authid = 'abcdef123';
serverip = '111.222.333.44';
portnumber = 8194;
terminalip = '5555.222.333.44';
c = emsx(servicename,authid,serverip,portnumber,terminalip)
```

```
c =

  emsx with properties:

      Session: [1x1 com.bloomberglp.blpapi.Session]
      Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
```

```
    Ipaddress: '111.222.333.44'
         Port: 8194
         User: [1×1 com.bloomberglp.blpapi.impl.by]
```

MATLAB returns `c` as the connection to the Bloomberg EMSX test service with the following properties:

- Bloomberg EMSX session object
- Bloomberg EMSX service object
- IP address of the machine running the Bloomberg EMSX test service
- Port number of the machine running the Bloomberg EMSX test service
- Bloomberg API Java object

Define the broker and strategy information structure `brokerstrat`. Obtain broker information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

The `EMSX_BROKERS` field lists the Bloomberg EMSX brokers.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
```

```
r = getBrokerInfo(c,brokerstrat)
```

```
 r =

        EMSX_BROKERS: {2x1 cell}
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Version History

Introduced in R2013a

See Also

Topics

“Create Order Using Bloomberg EMSX” on page 4-2
“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16

External Websites

EMSX API Programmers Guide

close

Close Bloomberg EMSX connection

Syntax

```
close(c)
```

Description

`close(c)` closes the Bloomberg EMSX connection `c`.

Examples

Close the Bloomberg EMSX Connection

Create the Bloomberg EMSX connection `c`.

```
c = emsx('//blp/emapisvc_beta');
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

Version History

Introduced in R2013a

See Also

`emsx` | `createOrder` | `routeOrder` | `createOrderAndRoute`

Topics

“Create Order Using Bloomberg EMSX” on page 4-2
“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16

External Websites

EMSX API Programmers Guide

createOrder

Create Bloomberg EMSX order

Syntax

```
events = createOrder(c,order)
events = createOrder(c,order,'timeOut',timeout)

createOrder( ___ ,'useDefaultEventHandler',false)

 ___  = createOrder(c,order,options)
```

Description

`events = createOrder(c,order)` creates a Bloomberg EMSX order using the Bloomberg EMSX connection `c` and order request `order` that contains the required fields for creating an order. `createOrder` returns the order sequence number and status message using the default event handler.

`events = createOrder(c,order,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`createOrder( ___ ,'useDefaultEventHandler',false)` creates a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with creating orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

` ___  = createOrder(c,order,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Create an Order Using the Default Event Handler

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
```

```
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order`.

```
events = createOrder(c,order)
```

```
events =

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create an Order Using a Timeout

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order`. Set the timeout value to 200 milliseconds.

```
events = createOrder(c,order,'timeOut',200)
```

```
events =
```

```
    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create Order Using Custom Event Handler

To create a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Create the order using the Bloomberg EMSX connection `c` and `order`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with creating an order.

```
createOrder(c,order,'useDefaultEventHandler',false)
```

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(subs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create an Order Using an Options Structure

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Create the order using the Bloomberg EMSX connection `c`, `order`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = createOrder(c,order,options)
```

```
events =

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

order — Order request

structure

Order request, specified as a structure using Bloomberg EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `order`. Convert the number of shares to a 32-bit signed integer using `int32`. `order` contains these fields.

Field	Description
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX amount of shares
`EMSX_ORDER_TYPE`	Bloomberg EMSX order type
`EMSX_BROKER`	Bloomberg EMSX broker name
`EMSX_TIF`	Bloomberg EMSX time in force
`EMSX_HAND_INSTRUCTION`	Bloomberg EMSX hand instruction
`EMSX_SIDE`	Bloomberg EMSX buy or sell specification

Example:
```
order.EMSX_TICKER = 'XYZ';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Data Types: `struct`

timeout — Timeout value

`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

options — Options for custom event handler or timeout value

structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2013a

See Also

`emsx` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `orders` | `modifyOrder` | `deleteOrder` | `deleteRoute` | `routes` | `routeOrder` | `close` | `timer` | `start` | `stop` | `delete` | `manualFill`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

createOrderAndRoute

Create and route Bloomberg EMSX order

Syntax

```
events = createOrderAndRoute(c,order)
events = createOrderAndRoute(c,order,'timeOut',timeout)

createOrderAndRoute( ___ ,'useDefaultEventHandler',false)

 ___  = createOrderAndRoute(c,order,options)
```

Description

`events = createOrderAndRoute(c,order)` creates and routes a Bloomberg EMSX order using Bloomberg EMSX connection `c` and order request `order`. `createOrderAndRoute` returns the order sequence number, route number, and status message using the default event handler.

`events = createOrderAndRoute(c,order,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`createOrderAndRoute( ___ ,'useDefaultEventHandler',false)` creates and routes a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with creating and routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

` ___  = createOrderAndRoute(c,order,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Create and Route an Order Using the Default Event Handler

To create and route a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
```

```
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection `c` and `order`.

```
events = createOrderAndRoute(c,order)
```

```
events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using a Timeout

To create and route a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection `c` and `order`. Set the timeout value to 200 milliseconds.

```
events = createOrderAndRoute(c,order,'timeOut',200)
```

```
events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using a Custom Event Handler

To create and route a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Create and route the order using the Bloomberg EMSX connection `c` and `order`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with creating and routing an order.

```
createOrderAndRoute(c,order,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using an Options Structure

To create and route a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Create and route the order using the Bloomberg EMSX connection `c`, `order`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = createOrderAndRoute(c,order,options)
```

```
events =

    EMSX_SEQUENCE: 728924
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number

- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`order` — Order request

structure

Order request, specified as a structure using Bloomberg EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `order`. Convert the number of shares to a 32-bit signed integer using `int32`. `order` contains these fields.

Field	Description
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX amount of shares
`EMSX_ORDER_TYPE`	Bloomberg EMSX order type
`EMSX_BROKER`	Bloomberg EMSX broker name
`EMSX_TIF`	Bloomberg EMSX time in force
`EMSX_HAND_INSTRUCTION`	Bloomberg EMSX hand instruction
`EMSX_SIDE`	Bloomberg EMSX buy or sell specification

Example: `order.EMSX_TICKER = 'XYZ';`

```
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Data Types: `struct`

`timeout` — Timeout value

`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2013a

See Also

`emsx` | `createOrder` | `createOrderAndRouteWithStrat` | `orders` | `deleteOrder` | `routes` | `routeOrder` | `modifyOrder` | `deleteRoute` | `close` | `timer` | `start` | `stop` | `delete`

Topics
“Create Order Using Bloomberg EMSX” on page 4-2
“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites
EMSX API Programmers Guide

createOrderAndRouteWithStrat

Create and route Bloomberg EMSX order with strategies

Syntax

```
events = createOrderAndRouteWithStrat(c,order,strat)
events = createOrderAndRouteWithStrat(c,order,strat,'timeOut',timeout)

createOrderAndRouteWithStrat(___,'useDefaultEventHandler',false)

___ = createOrderAndRouteWithStrat(c,order,strat,options)
```

Description

`events = createOrderAndRouteWithStrat(c,order,strat)` creates and routes a Bloomberg EMSX order with strategies using Bloomberg EMSX connection `c`, order request `order`, and order strategy `strat`. `createOrderAndRouteWithStrat` returns the order sequence number, route number, and status message using the default event handler.

`events = createOrderAndRouteWithStrat(c,order,strat,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`createOrderAndRouteWithStrat(___,'useDefaultEventHandler',false)` creates and routes a Bloomberg EMSX order with strategies using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with creating and routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___ = createOrderAndRouteWithStrat(c,order,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Create and Route an Order Using the Default Event Handler

To create and route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
```

```
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create and route the order with strategies using the Bloomberg EMSX connection `c`, `order`, and `strat`.

```
events = createOrderAndRouteWithStrat(c,order,strat)
```

```
events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using a Timeout

To create and route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
```

```
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create and route the order with strategies using the Bloomberg EMSX connection `c`, `order`, and `strat`. Set the timeout value to 200 milliseconds.

```
events = createOrderAndRouteWithStrat(c,order,strat,'timeOut',200)
```

```
events =

    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using a Custom Event Handler

To create and route a Bloomberg EMSX order with strategies, create the Bloomberg EMSX connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
```

```
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Create and route the order with strategies using the Bloomberg EMSX connection `c`, `order`, and `strat`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with creating and routing an order.

```
createOrderAndRouteWithStrat(c,order,strat,...
                             'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create and Route an Order Using an Options Structure

To create and route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
```

```
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Create and route the order using the Bloomberg EMSX connection `c`, `order`, `strat`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = createOrderAndRouteWithStrat(c,order,strat,options)
```

```
events =

    EMSX_SEQUENCE: 728924
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

The default event handler processes the events associated with creating and routing the order. `createOrderAndRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`order` — Order request

structure

Order request, specified as a structure using Bloomberg EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `order`. Convert the number of shares to a 32-bit signed integer using `int32`. `order` contains these fields.

Field	Description
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX amount of shares
`EMSX_ORDER_TYPE`	Bloomberg EMSX order type
`EMSX_BROKER`	Bloomberg EMSX broker name
`EMSX_TIF`	Bloomberg EMSX time in force
`EMSX_HAND_INSTRUCTION`	Bloomberg EMSX hand instruction
`EMSX_SIDE`	Bloomberg EMSX buy or sell specification

Example:
```
order.EMSX_TICKER = 'XYZ';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Data Types: `struct`

`strat` — Order strategies
structure

Order strategies, specified as a structure that contains the fields: `EMSX_STRATEGY_NAME`, `EMSX_STRATEGY_FIELD_INDICATORS`, and `EMSX_STRATEGY_FIELDS`. The structure field values must align with the strategy fields specified by `EMSX_STRATEGY_NAME`. For details about strategy fields and ordering, see `getBrokerInfo`.

Convert `EMSX_STRATEGY_FIELD_INDICATORS` to a 32-bit signed integer using `int32`. Set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `0` for each field to use the field data setting in `EMSX_FIELD_DATA`. Or, set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `1` to ignore the data in `EMSX_FIELD_DATA`.

Example:
```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Data Types: `struct`

`timeout` — Timeout value
`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2013a

See Also

`getBrokerInfo` | `createOrder` | `deleteOrder` | `routes` | `routeOrder` | `emsx` | `orders` | `modifyOrder` | `deleteRoute` | `close` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

deleteOrder

Delete Bloomberg EMSX order

Syntax

```
events = deleteOrder(c,ordernum)
events = deleteOrder(c,ordernum,'timeOut',timeout)

deleteOrder( ___ ,'useDefaultEventHandler',false)

 ___  = deleteOrder(c,ordernum,options)
```

Description

`events = deleteOrder(c,ordernum)` deletes a Bloomberg EMSX order using the Bloomberg EMSX connection `c` and order number or structure `ordernum`. `deleteOrder` returns a status message using the default event handler.

`events = deleteOrder(c,ordernum,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`deleteOrder( ___ ,'useDefaultEventHandler',false)` deletes a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with deleting orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___  = deleteOrder(c,ordernum,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Delete an Order Using the Default Event Handler

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Delete the order using the Bloomberg EMSX connection `c` and `ordernum`.

```
events = deleteOrder(c,ordernum)

events =

      STATUS: '0'
     MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting the order. `deleteOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete an Order Using the Order Number Integer

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Delete the order using the Bloomberg EMSX connection `c` and the order sequence number `335877` for the order to delete.

```
events = deleteOrder(c,335877)

events =

      STATUS: '0'
     MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting the order. `deleteOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete an Order Using a Timeout

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Delete the order using the Bloomberg EMSX connection `c` and `ordernum`. Set the timeout value to 200 milliseconds.

```
events = deleteOrder(c,ordernum,'timeOut',200)
```

```
events =

     STATUS: '0'
    MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting the order. `deleteOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete an Order Using a Custom Event Handler

To delete a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Delete the order using the Bloomberg EMSX connection `c` and `ordernum`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with deleting an order.

```
deleteOrder(c,ordernum,'useDefaultEventHandler',false)
```

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(subs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete an Order Using an Options Structure

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Delete the order using the Bloomberg EMSX connection `c`, `ordernum`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = deleteOrder(c,ordernum,options)
```

```
events =

     STATUS: '0'
    MESSAGE: 'Order deleted'
```

The default event handler processes the events associated with deleting the order. `deleteOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`ordernum` — Order numbers to delete

structure | integer

Order numbers to delete, specified as a structure or an integer to denote one or more order sequence numbers.

Data Types: `struct` | `int32`

`timeout` — Timeout value

`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value

structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2013a

See Also

`createOrderAndRoute` | `orders` | `createOrder` | `routes` | `modifyOrder` | `routeOrder` | `emsx` | `deleteRoute` | `close` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

deleteRoute

Delete Bloomberg EMSX active shares

Syntax

```
events = deleteRoute(c,routenum)
events = deleteRoute(c,routenum,'timeOut',timeout)

deleteRoute( ___ ,'useDefaultEventHandler',false)

 ___  = deleteRoute(c,routenum,options)
```

Description

`events = deleteRoute(c,routenum)` deletes the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and route number `routenum`. `deleteRoute` returns a status message using the default event handler.

`events = deleteRoute(c,routenum,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`deleteRoute( ___ ,'useDefaultEventHandler',false)` deletes the active shares that are routed but not filled using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with deleting the active shares. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___  = deleteRoute(c,routenum,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Delete Active Shares

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

1. Create the connection `c` using `emsx`.
2. Set up an order and route subscription using `orders` and `routes`.
3. Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see “Create and Manage Bloomberg EMSX Route” on page 4-8.

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and `routenum`.

```
events = deleteRoute(c,routenum)
```

```
events =

     STATUS: '1'
    MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting the active shares. `deleteRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete Active Shares Using a Timeout

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

1. Create the connection `c` using `emsx`.
2. Set up an order and route subscription using `orders` and `routes`.
3. Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see “Create and Manage Bloomberg EMSX Route” on page 4-8.

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and `routenum`. Set the timeout value to 200 milliseconds.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;
```

```
events = deleteRoute(c,routenum,'timeOut',200)

events =

     STATUS: '1'
    MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting the active shares. `deleteRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete Active Shares Using a Custom Event Handler

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

1. Create the Bloomberg EMSX connection `c` using `emsx`.
2. Set up an order and route subscription using `orders` and `routes`.
3. Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see “Create and Manage Bloomberg EMSX Route” on page 4-8.

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and `routenum`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with deleting the active shares.

```
deleteRoute(c,routenum,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Delete Active Shares Using an Options Structure

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

1. Create the connection `c` using `emsx`.
2. Set up an order and route subscription using `orders` and `routes`.
3. Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see “Create and Manage Bloomberg EMSX Route” on page 4-8.

Define the structure `routenum` that contains the order sequence number `EMSX_SEQUENCE` for the routed order and route number `EMSX_ROUTE_ID`.

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c`, `routenum`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = deleteRoute(c,routenum,options)
```

```
events =

     STATUS: '1'
    MESSAGE: 'Route cancellation request sent to broker'
```

The default event handler processes the events associated with deleting the active shares. `deleteRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX status

- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`routenum` — Route to delete

structure

Route to delete, specified as a structure containing fields `EMSX_SEQUENCE` and `EMSX_ROUTE_ID`.

Example: `routenum.EMSX_SEQUENCE = 728918;`
`routenum.EMSX_ROUTE_ID = 1;`

Data Types: `struct`

`timeout` — Timeout value

`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value

structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2013a

See Also

`createOrderAndRoute` | `orders` | `createOrder` | `routes` | `modifyRoute` | `deleteOrder` | `routeOrder` | `emsx` | `modifyOrder` | `close` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

emsxOrderBlotter

Bloomberg EMSX example order blotter

Syntax

```
[t,subs] = emsxOrderBlotter(c)
```

Description

`[t,subs] = emsxOrderBlotter(c)` displays a trader's order information. `c` is the Bloomberg EMSX connection, `t` is the timer object associated with the event handler, and `subs` is the Bloomberg EMSX subscription list.

Examples

Display the Order in an Order Blotter

Create the Bloomberg EMSX connection `c`.

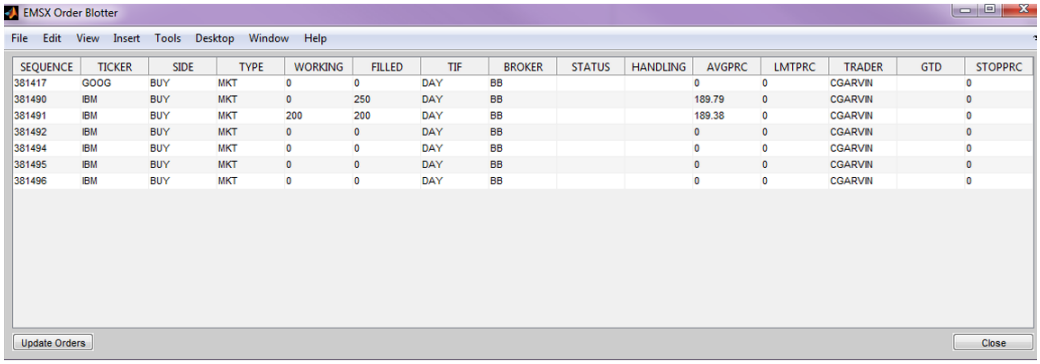
```
 c = emsx('//blp/emapisvc_beta');
```

Open Bloomberg EMSX order blotter using the Bloomberg EMSX connection `c`.

```
[t,subs] = emsxOrderBlotter(c)
```

```
   Timer Object: timer-1

   Timer Settings
      ExecutionMode: fixedRate
             Period: 1
           BusyMode: drop
            Running: on

   Callbacks
           TimerFcn: {@processEventToBlotter [1x1 emsx]}
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''


subs =

com.bloomberglp.blpapi.SubscriptionList@3e24da58
```

`emsxOrderBlotter` returns the timer object output and the Bloomberg EMSX subscription list object. For details about the timer object, see `timer`.



The order blotter displays the current order information for a trader.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 330 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

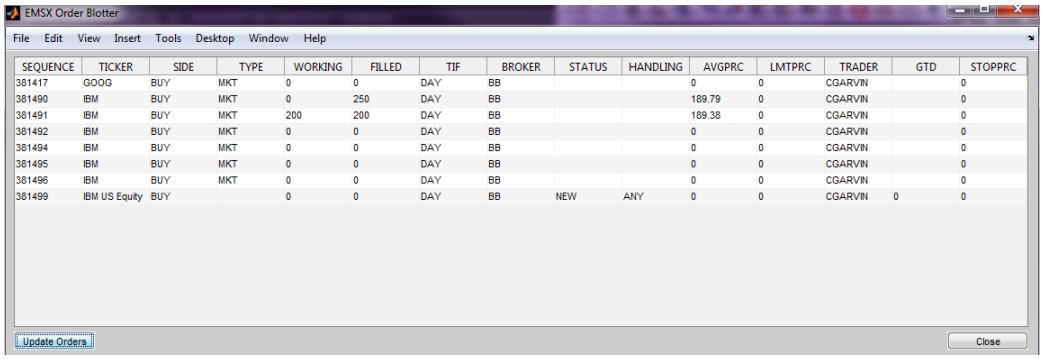
```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(330);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection `c` and the order request structure `order`. Use the custom event handler `processEventToBlotter` by setting the name-value pair argument `'useDefaultEventHandler'` to `false`.

```
events = createOrderAndRoute(c,order,'useDefaultEventHandler',false)
```

```
events =

     []

 CreateOrderAndRoute = {

     EMSX_SEQUENCE = 381499

     EMSX_ROUTE_ID = 1

     MESSAGE = Order created and routed

     }
```

`createOrderAndRoute` creates the order, routes the order, and returns a structure `events` that contains an empty double. `processEventToBlotter` displays output from `createOrderAndRoute` with the order number `EMSX_SEQUENCE`, route number `EMSX_ROUTE_ID`, and message: Order created and routed.



The order blotter updates using the information for the created and routed order, where order number `EMSX_SEQUENCE` is `381499`, using the event handler function `processEventToBlotter`. The order blotter updates as orders are created and managed.

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

Output Arguments

`t` — MATLAB timer

object

MATLAB timer, returned as a MATLAB object. For details, see `timer`.

`subs` — Bloomberg EMSX subscription list

subscription list object

Bloomberg EMSX subscription list, returned as a Bloomberg EMSX subscription list object.

Version History

Introduced in R2013a

See Also

`emsx` | `createOrder` | `timer` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `orders` | `modifyOrder` | `deleteOrder` | `deleteRoute` | `routes` | `routeOrder` | `createOrder` | `close`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12

“Workflow for Bloomberg EMSX” on page 4-16

External Websites
EMSX API Programmers Guide

getAllFieldMetaData

Obtain Bloomberg EMSX field information

Syntax

```
r = getAllFieldMetaData(c)
```

Description

`r = getAllFieldMetaData(c)` returns the Bloomberg EMSX field information using the Bloomberg EMSX connection `c`.

Examples

Request All Field Information

Create a connection `c` to the Bloomberg EMSX.

```
c = emsx('//blp/emapisvc_beta');
```

Request all fields supported by Bloomberg EMSX service using the Bloomberg EMSX connection `c`.

```
r = getAllFieldMetaData(c)
```

```
r =

       EMSX_FIELD_NAME: {113x1 cell}
        EMSX_DISP_NAME: {113x1 cell}
             EMSX_TYPE: {113x1 cell}
            EMSX_LEVEL: [113x1 double]
              EMSX_LEN: [113x1 double]
```

Display all field information for the first Bloomberg EMSX field using a cell array. Create a cell array from the fields in the returned data structure `r`.

```
{r.EMSX_FIELD_NAME{1} r.EMSX_DISP_NAME{1} r.EMSX_TYPE{1} r.EMSX_LEVEL(1) r.EMSX_LEN(1)}
```

```
 'MSG_TYPE'    'Msg Type'    'String'    [0]    [1]
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

Output Arguments

`r` — Return information for all fields
structure

Return information for all fields, returned as a structure for all fields supported by Bloomberg EMSX.

Version History

Introduced in R2013a

See Also

`emsx` | `close` | `createOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16

External Websites

EMSX API Programmers Guide

getBrokerInfo

Obtain Bloomberg EMSX broker and strategy information

Syntax

```
r = getBrokerInfo(c,brokerstrat)
```

Description

`r = getBrokerInfo(c,brokerstrat)` obtains Bloomberg EMSX broker and strategy information using the Bloomberg EMSX connection `c` and broker and strategy request structure `brokerstrat`.

Examples

Obtain Broker Information

Create a connection `c` to the Bloomberg EMSX.

```
 c = emsx('//blp/emapisvc_beta');
```

Define the broker and strategy information structure `brokerstrat`. Obtain broker information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';

r = getBrokerInfo(c,brokerstrat)
```

```
 r =

        EMSX_BROKERS: {2x1 cell}
```

The `EMSX_BROKERS` field lists the Bloomberg EMSX brokers.

Close the Bloomberg EMSX connection.

```
close(c)
```

Obtain Strategy Information

Create a connection `c` to the Bloomberg EMSX.

```
 c = emsx('//blp/emapisvc_beta');
```

Define the broker and strategy information structure `brokerstrat`. Obtain strategy information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
brokerstrat.EMSX_BROKER = 'BMTB';

r = getBrokerInfo(c,brokerstrat)
```

```
 r =

        EMSX_STRATEGIES: {16x1 cell}
```

The `EMSX_STRATEGIES` field lists the Bloomberg EMSX strategies.

Close the Bloomberg EMSX connection.

```
close(c)
```

Obtain Field Information

Create a connection `c` to the Bloomberg EMSX.

```
 c = emsx('//blp/emapisvc_beta');
```

Define the broker and strategy information structure `brokerstrat`. Obtain field information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
brokerstrat.EMSX_BROKER = 'BMTB';
brokerstrat.EMSX_STRATEGY = 'SSP';

r = getBrokerInfo(c,brokerstrat)

r =

        FieldName: {3x1 cell}
        Disable: {3x1 cell}
        StringValue: {3x1 cell}
```

The structure field `FieldName` lists the Bloomberg EMSX fields. The structure fields `Disable` and `StringValue` contain information about the Bloomberg EMSX fields.

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`brokerstrat` — Broker and strategy request

structure

Broker and strategy request, specified as a structure that contains Bloomberg EMSX fields. Use `getAllFieldMetaData` to view all available fields for `brokerStrategyStruct`.

Example: `brokerstrat.EMSX_TICKER = 'ABCD US Equity';`

Data Types: `struct`

Output Arguments

`r` — Broker and strategy information
structure

Broker and strategy information, returned as a structure.

Version History

Introduced in R2013a

See Also

`createOrder` | `createOrderAndRoute` | `orders` | `modifyOrder` | `routes` | `deleteOrder` | `emsx` | `createOrderAndRouteWithStrat` | `deleteRoute` | `routeOrder` | `close`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16

External Websites

EMSX API Programmers Guide

groupRouteOrderWithStrat

Route group of Bloomberg EMSX orders with strategies

Syntax

```
events = groupRouteOrderWithStrat(c,route,strat)
events = groupRouteOrderWithStrat(c,route,strat,'timeOut',timeout)

groupRouteOrderWithStrat( ___ ,'useDefaultEventHandler',false)

 ___  = groupRouteOrderWithStrat(c,route,strat,options)
```

Description

`events = groupRouteOrderWithStrat(c,route,strat)` routes multiple Bloomberg EMSX orders with strategies using the Bloomberg EMSX connection `c`, route request `route`, and strategy `strat`. `routeOrderWithStrat` returns the order sequence number, route number, and status message using the default event handler.

`events = groupRouteOrderWithStrat(c,route,strat,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`groupRouteOrderWithStrat( ___ ,'useDefaultEventHandler',false)` routes multiple Bloomberg EMSX orders with strategies using any of the input arguments in the previous syntaxes and a custom event handler. To process the events associated with routing orders, write a custom event handler. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

` ___  = groupRouteOrderWithStrat(c,route,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true`, and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Route Orders Using the Default Event Handler

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies these route request fields:

- Order numbers 335877 and 335878
- Stock symbol IBM
- 100 percent of shares shown on the order to be routed
- Broker BMTB
- Any hand instruction
- Time in force set to DAY
- Market order type

```
route.EMSX_SEQUENCE = {int32(335877);int32(335878)};
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT_PERCENT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Create the order strategies structure `strat` using the strategy SSP. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Route the orders using the Bloomberg EMSX connection `c`, `route`, and `strat`.

```
events = groupRouteOrderWithStrat(c,route,strat)
```

```
events =

     EMSX_SUCCESS_ROUTES: [1x1 struct]
      EMSX_FAILED_ROUTES: [1x1 struct]
                 MESSAGE: '1 of 1 Order(s) Routed'

   where

   events.EMSX_SUCCESS_ROUTES =

        EMSX_SEQUENCE: 335877
        EMSX_ROUTE_ID: 1

   and events.EMSX_FAILED_ROUTES =

        EMSX_SEQUENCE: 335878
           ERROR_CODE: 0
        ERROR_MESSAGE: {'Order 335878  View-only orders can not be routed'}
```

The default event handler processes the events associated with routing the order. `groupRouteOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX success routing structure, which contains the order number and route identifier for the orders that successfully routed
- Bloomberg EMSX failed routing structure, which contains the order number, error code, and error message for the orders that failed to route
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Orders Using a Timeout

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies these route request fields:

- Order numbers `335877` and `335878`
- Stock symbol IBM
- 100 percent of shares shown on the order to be routed
- Broker `BMTB`
- Any hand instruction
- Time in force set to `DAY`
- Market order type

```
route.EMSX_SEQUENCE = {int32(335877);int32(335878)};
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT_PERCENT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Route the orders using the Bloomberg EMSX connection `c`, `route`, and `strat`. Set the timeout value to 200 milliseconds.

```
events = groupRouteOrderWithStrat(c,route,strat,'timeOut',200)
```

```
events =

     EMSX_SUCCESS_ROUTES: [1x1 struct]
      EMSX_FAILED_ROUTES: [1x1 struct]
                 MESSAGE: '1 of 1 Order(s) Routed'

   where

   events.EMSX_SUCCESS_ROUTES =

        EMSX_SEQUENCE: 335877
        EMSX_ROUTE_ID: 1

   and events.EMSX_FAILED_ROUTES =

        EMSX_SEQUENCE: 335878
```

```
    ERROR_CODE: 0
ERROR_MESSAGE: {'Order 335878  View-only orders can not be routed'}
```

The default event handler processes the events associated with routing the order. `groupRouteOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX success routing structure, which contains the order number and route identifier for the orders that successfully routed
- Bloomberg EMSX failed routing structure, which contains the order number, error code, and error message for the orders that failed to route
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Orders Using a Custom Event Handler

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies these route request fields:

- Order numbers `335877` and `335878`
- Stock symbol IBM
- 100 percent of shares shown on the order to be routed
- Broker `BMTB`
- Any hand instruction
- Time in force set to `DAY`
- Market order type

```
route.EMSX_SEQUENCE = {int32(335877);int32(335878)};
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT_PERCENT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Suppose that you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. To run `eventhandler` immediately, start the timer using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Route the orders using the Bloomberg EMSX connection `c`, `route`, and `strat`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with routing an order.

```
groupRouteOrderWithStrat(c,route,strat,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. To stop data updates, stop the timer using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Orders Using an Options Structure

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies these route request fields:

- Order numbers `335877` and `335878`
- Stock symbol IBM
- 100 percent of shares shown on the order to be routed
- Broker `BMTB`
- Any hand instruction
- Time in force set to `DAY`

- Market order type

```
route.EMSX_SEQUENCE = {int32(335877);int32(335878)};
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT_PERCENT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Route the orders using the Bloomberg EMSX connection `c`, `route`, `strat`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = groupRouteOrderWithStrat(c,route,strat,options)
```

```
events =

     EMSX_SUCCESS_ROUTES: [1x1 struct]
      EMSX_FAILED_ROUTES: [1x1 struct]
                 MESSAGE: '1 of 1 Order(s) Routed'

   where

   events.EMSX_SUCCESS_ROUTES =

        EMSX_SEQUENCE: 335877
        EMSX_ROUTE_ID: 1

   and events.EMSX_FAILED_ROUTES =

        EMSX_SEQUENCE: 335878
           ERROR_CODE: 0
        ERROR_MESSAGE: {'Order 335878  View-only orders can not be routed'}
```

The default event handler processes the events associated with routing the order. `groupRouteOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX success routing structure, which contains the order number and route identifier for the orders that successfully routed
- Bloomberg EMSX failed routing structure, which contains the order number, error code, and error message for the orders that failed to route
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

route — Route request

structure

Route request, specified as a structure containing these fields.

Convert the numbers to 32-bit signed integers using `int32`. `EMSX_SEQUENCE` must denote an existing order sequence number.

Field	Description
`EMSX_SEQUENCE`	Bloomberg EMSX order sequence number
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX number of shares
`EMSX_BROKER`	Bloomberg EMSX broker name
`EMSX_HAND_INSTRUCTION`	Bloomberg EMSX hand instruction
`EMSX_TIF`	Bloomberg EMSX time in force
`EMSX_ORDER_TYPE`	Bloomberg EMSX order type

Example:
```
route.EMSX_SEQUENCE = int32(728918);
route.EMSX_TICKER = 'XYZ';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
route.EMSX_TIF = 'DAY';
route.EMSX_ORDER_TYPE = 'MKT';
```

Data Types: `struct`

strat — Order strategies

structure

Order strategies, specified as a structure that contains the fields: `EMSX_STRATEGY_NAME`, `EMSX_STRATEGY_FIELD_INDICATORS`, and `EMSX_STRATEGY_FIELDS`. The structure field values must align with the strategy fields specified by `EMSX_STRATEGY_NAME`. For details about strategy fields and ordering, see `getBrokerInfo`.

Convert `EMSX_STRATEGY_FIELD_INDICATORS` to a 32-bit signed integer using `int32`. Set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `0` for each field to use the field data setting in `EMSX_FIELD_DATA`. Or, set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `1` to ignore the data in `EMSX_FIELD_DATA`.

Example:
```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Data Types: `struct`

`timeout` — Timeout value
`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2015b

See Also

`getBrokerInfo` | `createOrderAndRouteWithStrat` | `createOrder` | `orders` | `deleteOrder` | `routes` | `routeOrder` | `routeOrderWithStrat` | `emsx` | `createOrderAndRoute` | `modifyOrder` | `deleteRoute` | `close` | `timer` | `start` | `stop` | `delete`

Topics
“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites
EMSX API Programmers Guide

modifyOrder

Modify Bloomberg EMSX order

Syntax

```
events = modifyOrder(c,modorder)
events = modifyOrder(c,modorder,'timeOut',timeout)

modifyOrder( ___ ,'useDefaultEventHandler',false)

 ___  = modifyOrder(c,modorder,options)
```

Description

`events = modifyOrder(c,modorder)` modifies a Bloomberg EMSX order using the Bloomberg EMSX connection `c` and modify order request structure `modorder`. `modifyOrder` returns a status message using the default event handler.

`events = modifyOrder(c,modorder,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`modifyOrder( ___ ,'useDefaultEventHandler',false)` modifies a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with modifying orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___  = modifyOrder(c,modorder,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Modify an Order Using the Default Event Handler

To modify a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number `728905` for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Modify the order using the Bloomberg EMSX connection `c` and `modorder`.

```
events = modifyOrder(c,modorder)
```

```
events =

    EMSX_SEQUENCE: 728905
          MESSAGE: 'Order Modified'
```

The default event handler processes the events associated with modifying the order. `modifyOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify an Order Using a Timeout

To modify a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number `728905` for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Modify the order using the Bloomberg EMSX connection `c` and `modorder`. Set the timeout value to 200 milliseconds.

```
events = modifyOrder(c,modorder,'timeOut',200)
```

```
events =

    EMSX_SEQUENCE: 728905
          MESSAGE: 'Order Modified'
```

The default event handler processes the events associated with modifying the order. `modifyOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify an Order Using a Custom Event Handler

To modify a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number `728905` for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Modify the order using the Bloomberg EMSX connection `c` and `modorder`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with modifying an order.

```
modifyOrder(c,modorder,'useDefaultEventHandler',false)
```

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(subs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify an Order Using an Options Structure

To modify a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number `728905` for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Modify the order using the Bloomberg EMSX connection `c`, `modorder`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = modifyOrder(c,modorder,options)
```

```
events =

    EMSX_SEQUENCE: 728905
          MESSAGE: 'Order Modified'
```

The default event handler processes the events associated with modifying the order. `modifyOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`modorder` — Modify order request

structure

Modify order request, specified as a structure that contains these fields.

Use `getAllFieldMetaData` to view all available fields for `modorder`. Convert the numbers to 32-bit signed integers using `int32`.

Field	Description
`EMSX_SEQUENCE`	Bloomberg EMSX order sequence number
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX number of shares

Example:
```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'XYZ';
modorder.EMSX_AMOUNT = int32(100);
```

Data Types: `struct`

`timeout` — Timeout value
`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2013a

See Also

`createOrderAndRoute` | `orders` | `createOrder` | `routes` | `deleteOrder` | `emsx` | `createOrderAndRouteWithStrat` | `deleteRoute` | `routeOrder` | `close` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

modifyRoute

Modify Bloomberg EMSX route

Syntax

```
events = modifyRoute(c,modroute)
events = modifyRoute(c,modroute,'timeOut',timeout)

modifyRoute( ___ ,'useDefaultEventHandler',false)

 ___  = modifyRoute(c,modroute,options)
```

Description

`events = modifyRoute(c,modroute)` modifies a Bloomberg EMSX route using the Bloomberg EMSX connection `c` and route request `modroute`. `modifyRoute` returns a status message using the default event handler.

`events = modifyRoute(c,modroute,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`modifyRoute( ___ ,'useDefaultEventHandler',false)` modifies a Bloomberg EMSX route using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with modifying routes. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___  = modifyRoute(c,modroute,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Modify a Route Using the Default Event Handler

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code instructs Bloomberg EMSX to route 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Modify the route using the Bloomberg EMSX connection `c` and `modroute`.

```
events = modifyRoute(c,modroute)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route Using a Timeout

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Modify the route using the Bloomberg EMSX connection `c` and `modroute`. Set the timeout value to 200 milliseconds.

```
events = modifyRoute(c,modroute,'timeOut',200)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route Using a Custom Event Handler

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Modify the route using the Bloomberg EMSX connection `c` and `modroute`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with modifying a route.

```
modifyRoute(c,modroute,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route Using an Options Structure

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = modifyRoute(c,modroute,options)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRoute` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`modroute` — Modify route request
structure

Modify route request, specified as a structure with these fields.

Use `getAllFieldMetaData` to view all available fields for `modroute`. Convert the numbers to 32-bit signed integers using `int32`.

Field	Description
`EMSX_SEQUENCE`	Bloomberg EMSX order sequence number
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX number of shares
`EMSX_ROUTE_ID`	Bloomberg EMSX route identifier

Example:
```
modroute.EMSX_SEQUENCE = int32(731128);
modroute.EMSX_TICKER = 'XYZ';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Data Types: `struct`

`timeout` — Timeout value
`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2013a

See Also

`createOrderAndRoute` | `orders` | `createOrder` | `routes` | `deleteOrder` | `modifyRouteWithStrat` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

modifyRouteWithStrat

Modify Bloomberg EMSX route with strategies

Syntax

```
events = modifyRouteWithStrat(c,modroute,strat)
events = modifyRouteWithStrat(c,modroute,strat,'timeOut',timeout)

modifyRouteWithStrat( ___ ,'useDefaultEventHandler',false)

 ___  = modifyRouteWithStrat(c,modroute,strat,options)
```

Description

`events = modifyRouteWithStrat(c,modroute,strat)` modifies a Bloomberg EMSX route with strategies using the Bloomberg EMSX connection `c`, route request `modroute`, and order strategy `strat`. `modifyRouteWithStrat` returns the order sequence number, route identifier, and status message using the default event handler.

`events = modifyRouteWithStrat(c,modroute,strat,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`modifyRouteWithStrat( ___ ,'useDefaultEventHandler',false)` modifies a Bloomberg EMSX route with strategies using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with modifying routes. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

` ___  = modifyRouteWithStrat(c,modroute,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Modify a Route with Strategies Using the Default Event Handler

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and `strat`.

```
events = modifyRouteWithStrat(c,modroute,strat)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route with Strategies Using a Timeout

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.

- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and `strat`. Set the timeout value to 200 milliseconds.

```
events = modifyRouteWithStrat(c,modroute,strat,'timeOut',200)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route with Strategies Using a Custom Event Handler

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and `strat`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with modifying a route.

```
modifyRouteWithStrat(c,modroute,strat,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Modify a Route with Strategies Using an Options Structure

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage Bloomberg EMSX Order and Route” on page 4-12.

Define the `modroute` structure that contains these fields:

- Bloomberg EMSX order sequence number `EMSX_SEQUENCE`
- Bloomberg EMSX ticker symbol `EMSX_TICKER`
- Bloomberg EMSX number of shares `EMSX_AMOUNT`
- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

This code modifies the route to 100 shares of IBM for order sequence number `731128` and route identifier `1`. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Modify the route using the Bloomberg EMSX connection `c`, `modroute`, `strat`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = modifyRouteWithStrat(c,modroute,strat,options)
```

```
events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
          MESSAGE: 'Route modified'
```

The default event handler processes the events associated with modifying a route. `modifyRouteWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`modroute` — Modify route request

structure

Modify route request, specified as a structure with these fields.

Use `getAllFieldMetaData` to view all available fields for `modroute`. Convert the numbers to 32-bit signed integers using `int32`.

Field	Description
`EMSX_SEQUENCE`	Bloomberg EMSX order sequence number
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX number of shares
`EMSX_ROUTE_ID`	Bloomberg EMSX route identifier

Example: `modroute.EMSX_SEQUENCE = int32(731128);`

```
modroute.EMSX_TICKER = 'XYZ';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Data Types: `struct`

`strat` — Order strategies

structure

Order strategies, specified as a structure that contains the fields: `EMSX_STRATEGY_NAME`, `EMSX_STRATEGY_FIELD_INDICATORS`, and `EMSX_STRATEGY_FIELDS`. The structure field values must align with the strategy fields specified by `EMSX_STRATEGY_NAME`. For details about strategy fields and ordering, see `getBrokerInfo`.

Convert `EMSX_STRATEGY_FIELD_INDICATORS` to a 32-bit signed integer using `int32`. Set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `0` for each field to use the field data setting in `EMSX_FIELD_DATA`. Or, set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `1` to ignore the data in `EMSX_FIELD_DATA`.

Example:
```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Data Types: `struct`

`timeout` — Timeout value
`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2013a

See Also

`getBrokerInfo` | `createOrderAndRouteWithStrat` | `createOrder` | `modifyRoute` | `orders` | `deleteOrder` | `routes` | `routeOrder` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

orders

Obtain Bloomberg EMSX order subscription

Syntax

```
[events,subs] = orders(c,fields)

[events,subs] = orders(c,fields,Name,Value)
[events,subs] = orders(c,fields,options)
```

Description

`[events,subs] = orders(c,fields)` subscribes to Bloomberg EMSX fields `fields` using the Bloomberg EMSX connection `c`. `orders` returns existing event data `events` from the event queue and the Bloomberg EMSX subscription list `subs`.

`[events,subs] = orders(c,fields,Name,Value)` uses additional options specified by one or more `Name`,`Value` pair arguments to specify a custom event handler or timeout value for the event handler.

`[events,subs] = orders(c,fields,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The `options` structure fields and values correspond to names and values of name-value pair arguments, respectively.

Examples

Subscribe to Order Events Using the Default Event Handler

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c` and Bloomberg EMSX field list `fields`.

```
fields = {'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'};

[events,subs] = orders(c,fields)
```

```
events =

                    MSG_TYPE: {'E'}
                MSG_SUB_TYPE: {'O'}
                EVENT_STATUS: 4
                ...

subs =

com.bloomberglp.blpapi.SubscriptionList@4bc3dc78
```

`events` contains fields for the events currently in the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Subscribe to Order Events Using the Custom Event Handler

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Write a custom event handler function named `eventhandler`. Run the custom event handler using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate');
start(t)
```

`t` is the timer object.

Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c` and Bloomberg EMSX field list `fields`. Use the custom event handler by setting the name-value pair argument `'useDefaultEventHandler'` to `false`.

```
fields = {'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'};

[events,subs] = orders(c,fields,'useDefaultEventHandler',false)

events =

     []


subs =

com.bloomberglp.blpapi.SubscriptionList@2c5b1c7e
```

`events` contains an empty double. The custom event handler processes the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(subs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Subscribe to Order Events Using a Timeout

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c` and Bloomberg EMSX field list `fields`. Specify the name-value pair argument `timeOut` and set it to 200 milliseconds.

```
fields = {'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'};

[events,subs] = orders(c,fields,'timeOut',200)

events =

                    MSG_TYPE: {'E'}
                MSG_SUB_TYPE: {'O'}
                EVENT_STATUS: 4
                ...

subs =

com.bloomberglp.blpapi.SubscriptionList@4bc3dc78
```

`events` contains fields for the events currently in the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Subscribe to Order Events Using the Options Structure

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c`, Bloomberg EMSX field list `fields`, and options structure `options`.

```
options.timeOut = 200;
options.useDefaultEventHandler = true;

fields = {'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'};

[events,subs] = orders(c,fields,options)

events =
```

```
                    MSG_TYPE: {'E'}
                MSG_SUB_TYPE: {'O'}
                EVENT_STATUS: 4
                ...

subs =

com.bloomberglp.blpapi.SubscriptionList@4bc3dc78
```

`events` contains fields for the events currently in the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`fields` — Bloomberg EMSX field information

cell array

Bloomberg EMSX field information, specified using a cell array. Use `getAllFieldMetaData` to view available field information for the Bloomberg EMSX service.

Example: `'EMSX_TICKER'`
`'EMSX_AMOUNT'`
`'EMSX_ORDER_TYPE'`

Data Types: `cell`

`options` — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure instead of name-value pair arguments to reuse the optional name-value pair arguments to specify a custom event handler or timeout value for the event handler.

The `options` structure field and values correspond to names and values of the name-value pair arguments, respectively.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: `options.useDefaultEventHandler = false;`
`options.timeOut = 500;`

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'useDefaultEventHandler',false`

`useDefaultEventHandler` — Flag for event handler preference
`true` (default) | `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, specified as the comma-separated pair consisting of `'useDefaultEventHandler'` and the logical values `true` or `false`.

To specify the default event handler, set this flag to `true`.

Otherwise, set this flag to `false` to specify a custom event handler.

Data Types: `logical`

`timeOut` — Timeout value for event handler
`500` milliseconds (default) | nonnegative integer

Timeout value for event handler for the Bloomberg EMSX service, specified as the comma-separated pair consisting of `'timeOut'` and a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types: `double`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

When the name-value pair argument `'useDefaultEventHandler'` or the same field for the structure `options` is set to `false`, `events` is an empty double.

`subs` — Bloomberg EMSX subscription list
subscription list object

Bloomberg EMSX subscription list, returned as a Bloomberg EMSX subscription list object.

Version History

Introduced in R2013a

See Also

`emsx` | `createOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `modifyOrder` | `deleteOrder` | `deleteRoute` | `routes` | `routeOrder` | `getAllFieldMetaData` | `close` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

processEvent

Sample Bloomberg EMSX event handler

Syntax

```
processEvent(c)
```

Description

`processEvent(c)` displays and flushes the event queue associated with the Bloomberg EMSX connection `c`. `processEvent` is a sample event handler function. You can build a custom event handler function to process Bloomberg EMSX events.

Examples

Continually Process Bloomberg EMSX Event Queue

Create the Bloomberg EMSX connection `c`.

```
c = emsx('//blp/emapisvc_beta');
```

Use `timer` to continually process the Bloomberg EMSX event queue.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
```

`t` is the MATLAB timer object. For details, see `timer`.

Close the Bloomberg EMSX connection.

```
close(c)
```

Process Bloomberg EMSX Event Queue Once

Create the Bloomberg EMSX connection `c`.

```
c = emsx('//blp/emapisvc_beta');
```

Use the default event handler function `processEvent` to process the Bloomberg EMSX event queue once.

```
processEvent(c)
```

```
SessionConnectionUp = {

    server = "localhost/127.0.0.1:8194"

}

SessionStarted = {
```

```
}

ServiceOpened = {

    serviceName = "//blp/emapisvc_beta"

}
```

`processEvent` clears the Bloomberg EMSX event queue.

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

Version History

Introduced in R2013a

See Also

`createOrderAndRoute` | `orders` | `modifyOrder` | `routes` | `deleteOrder` | `routeOrder` | `timer` | `emsx` | `createOrderAndRouteWithStrat` | `deleteRoute` | `createOrder` | `close`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

routeOrder

Route Bloomberg EMSX order

Syntax

```
events = routeOrder(c,route)
events = routeOrder(c,route,'timeOut',timeout)

routeOrder( ___ ,'useDefaultEventHandler',false)

 ___  = routeOrder(c,route,options)
```

Description

`events = routeOrder(c,route)` routes a Bloomberg EMSX order using the Bloomberg EMSX connection `c` and route request `route`. `routeOrder` returns a status message using the default event handler.

`events = routeOrder(c,route,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`routeOrder( ___ ,'useDefaultEventHandler',false)` routes a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___  = routeOrder(c,route,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Route an Order Using the Default Event Handler

To route a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
```

```
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Route the order using the Bloomberg EMSX connection `c` and `route`.

```
events = routeOrder(c,route)
```

```
events =

     EMSX_SEQUENCE: 335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using a Timeout

To route a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Route the order using the Bloomberg EMSX connection `c` and `route`. Set the timeout value to 200 milliseconds.

```
events = routeOrder(c,route,'timeOut',200)
```

```
events =
```

```
    EMSX_SEQUENCE: 335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using a Custom Event Handler

To route a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Route the order using the Bloomberg EMSX connection `c` and `route`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with routing an order.

```
routeOrder(c,route,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using an Options Structure

To route a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Route the order using the Bloomberg EMSX connection `c`, `route`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = routeOrder(c,route,options)
```

```
events =

     EMSX_SEQUENCE: 335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrder` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`route` — Route request

structure

Route request, specified as a structure containing these fields.

Convert the numbers to 32-bit signed integers using `int32`. `EMSX_SEQUENCE` must denote an existing order sequence number.

Field	Description
`EMSX_SEQUENCE`	Bloomberg EMSX order sequence number
`EMSX_TICKER`	Bloomberg EMSX ticker symbol
`EMSX_AMOUNT`	Bloomberg EMSX number of shares
`EMSX_BROKER`	Bloomberg EMSX broker name
`EMSX_HAND_INSTRUCTION`	Bloomberg EMSX hand instruction

Example:
```
route.EMSX_SEQUENCE = int32(728918);
route.EMSX_TICKER = 'XYZ';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Data Types: `struct`

`timeout` — Timeout value

`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value

structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2013a

See Also

`createOrder` | `createOrderAndRoute` | `orders` | `modifyOrder` | `routes` | `deleteOrder` | `emsx` | `createOrderAndRouteWithStrat` | `routeOrderWithStrat` | `deleteRoute` | `close` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

routeOrderWithStrat

Route Bloomberg EMSX order with strategies

Syntax

```
events = routeOrderWithStrat(c,route,strat)
events = routeOrderWithStrat(c,route,strat,'timeOut',timeout)

routeOrderWithStrat( ___ ,'useDefaultEventHandler',false)

 ___  = routeOrderWithStrat(c,route,strat,options)
```

Description

`events = routeOrderWithStrat(c,route,strat)` routes a Bloomberg EMSX order with strategies using the Bloomberg EMSX connection `c`, route request `route`, and strategy `strat`. `routeOrderWithStrat` returns the order sequence number, route number, and status message using the default event handler.

`events = routeOrderWithStrat(c,route,strat,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`routeOrderWithStrat( ___ ,'useDefaultEventHandler',false)` routes a Bloomberg EMSX order with strategies using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`___ = routeOrderWithStrat(c,route,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

Examples

Route an Order Using the Default Event Handler

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Route the order using the Bloomberg EMSX connection `c`, `route`, and `strat`.

```
events = routeOrderWithStrat(c,route,strat)
```

```
events =

     EMSX_SEQUENCE: 335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using a Timeout

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
```

```
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Route the order using the Bloomberg EMSX connection `c`, `route`, and `strat`. Set the timeout value to 200 milliseconds.

```
events = routeOrderWithStrat(c,route,strat,'timeOut',200)
```

```
events =

     EMSX_SEQUENCE: 335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using a Custom Event Handler

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
start(t)
```

`t` is the MATLAB timer object. For details, see `timer`.

Route the order using the Bloomberg EMSX connection `c`, `route`, and `strat`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with routing an order.

```
routeOrderWithStrat(c,route,strat,'useDefaultEventHandler',false)
```

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route an Order Using an Options Structure

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Route the order using the Bloomberg EMSX connection `c`, `route`, `strat`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = routeOrderWithStrat(c,route,strat,options)
```

```
events =

     EMSX_SEQUENCE: 335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing the order. `routeOrderWithStrat` returns `events` as a structure that contains these fields:

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier
- Bloomberg EMSX message

Unsubscribe from order and route events using the Bloomberg EMSX subscription list objects `osubs` and `rsubs`. This code assumes that `orders` creates `osubs` and `routes` creates `rsubs`.

```
c.Session.unsubscribe(osubs)
c.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`route` — Route request

structure

Route request, specified as a structure containing these fields.

Convert the numbers to 32-bit signed integers using `int32`. `EMSX_SEQUENCE` must denote an existing order sequence number.

Field	Description
EMSX_SEQUENCE	Bloomberg EMSX order sequence number
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX number of shares
EMSX_BROKER	Bloomberg EMSX broker name
EMSX_HAND_INSTRUCTION	Bloomberg EMSX hand instruction

Example:
```
route.EMSX_SEQUENCE = int32(728918);
route.EMSX_TICKER = 'XYZ';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Data Types: `struct`

`strat` — Order strategies
structure

Order strategies, specified as a structure that contains the fields: `EMSX_STRATEGY_NAME`, `EMSX_STRATEGY_FIELD_INDICATORS`, and `EMSX_STRATEGY_FIELDS`. The structure field values must align with the strategy fields specified by `EMSX_STRATEGY_NAME`. For details about strategy fields and ordering, see `getBrokerInfo`.

Convert `EMSX_STRATEGY_FIELD_INDICATORS` to a 32-bit signed integer using `int32`. Set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `0` for each field to use the field data setting in `EMSX_FIELD_DATA`. Or, set `EMSX_STRATEGY_FIELD_INDICATORS` equal to `1` to ignore the data in `EMSX_FIELD_DATA`.

Example:
```
strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
```

Data Types: `struct`

`timeout` — Timeout value
`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2013a

See Also

`getBrokerInfo` | `createOrderAndRouteWithStrat` | `createOrder` | `orders` | `deleteOrder` | `routes` | `routeOrder` | `emsx` | `createOrderAndRoute` | `modifyOrder` | `deleteRoute` | `close` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

routes

Obtain Bloomberg EMSX route subscription

Syntax

```
[events,subs] = routes(c,fields)

[events,subs] = routes(c,fields,Name,Value)
[events,subs] = routes(c,fields,options)
```

Description

`[events,subs] = routes(c,fields)` subscribes to Bloomberg EMSX fields `fields` using the Bloomberg EMSX connection `c`. `routes` returns existing event data `events` from the event queue and the Bloomberg EMSX subscription list `subs`.

`[events,subs] = routes(c,fields,Name,Value)` uses additional options specified by one or more `Name`,`Value` pair arguments to specify a custom event handler or timeout value for the event handler.

`[events,subs] = routes(c,fields,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The `options` structure fields and values correspond to names and values of name-value pair arguments, respectively.

Examples

Set Up Route Subscription Using the Default Event Handler

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};

[events,subs] = routes(c,fields)
```

```
events =

                    MSG_TYPE: {5x1 cell}
                MSG_SUB_TYPE: {5x1 cell}
                EVENT_STATUS: [5x1 int32]
                ...

subs =

com.bloomberglp.blpapi.SubscriptionList@463b9287
```

`events` contains fields for the events currently in the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from route events using the Bloomberg EMSX subscription list object `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Set Up Route Subscription Using a Custom Event Handler

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Write a custom event handler function named `eventhandler`. Run the custom event handler using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate');
start(t)
```

`t` is the timer object.

Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`. Use the custom event handler by setting the name-value pair argument `'useDefaultEventHandler'` to `false`.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};

[events,subs] = routes(c,fields,'useDefaultEventHandler',false)

events =

     []

subs =

com.bloomberglp.blpapi.SubscriptionList@463b9287
```

`events` is an empty double. The custom event handler processes the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from route events using the Bloomberg EMSX subscription list object `subs`. Stop the timer to stop data updates using `stop`.

```
c.Session.unsubscribe(subs)
stop(t)
```

If you are done processing data updates, delete the timer using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Set Up Route Subscription Using a Timeout

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`. Specify the name-value pair argument `timeOut` and set it to 200 milliseconds.

```
fields = {'EMSX_BROKER','EMSX_WORKING'};

[events,subs] = routes(c,fields,'timeOut',200)
```

```
events =

                    MSG_TYPE: {5x1 cell}
                MSG_SUB_TYPE: {5x1 cell}
                EVENT_STATUS: [5x1 int32]
                ...

subs =

com.bloomberglp.blpapi.SubscriptionList@463b9287
```

`events` contains fields for the events currently in the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from route events using the Bloomberg EMSX subscription list object `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Set Up Route Subscription Using an Options Structure

Create the Bloomberg EMSX connection `c`.

```
 c = emsx('//blp/emapisvc_beta');
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c` and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;
fields = {'EMSX_BROKER','EMSX_WORKING'};

[events,subs] = routes(c,fields,options)
```

```
events =

                    MSG_TYPE: {5x1 cell}
                MSG_SUB_TYPE: {5x1 cell}
                EVENT_STATUS: [5x1 int32]
                ...

subs =

com.bloomberglp.blpapi.SubscriptionList@463b9287
```

`events` contains fields for the events currently in the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from route events using the Bloomberg EMSX subscription list object `subs`.

```
c.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`fields` — Bloomberg EMSX field information

cell array

Bloomberg EMSX field information, specified using a cell array. Use `getAllFieldMetaData` to view available field information for the Bloomberg EMSX service.

Example: `'EMSX_TICKER'`
`'EMSX_AMOUNT'`
`'EMSX_ORDER_TYPE'`

Data Types: `cell`

`options` — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure instead of name-value pair arguments to reuse the optional name-value pair arguments to specify a custom event handler or timeout value for the event handler.

The `options` structure field and values correspond to names and values of the name-value pair arguments, respectively.

Specify using a custom event handler and a timeout value of 500 milliseconds.

Example: `options.useDefaultEventHandler = false;`
`options.timeOut = 500;`

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'useDefaultEventHandler',false`

`useDefaultEventHandler` — Flag for event handler preference
`true` (default) | `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, specified as the comma-separated pair consisting of `'useDefaultEventHandler'` and the logical values `true` or `false`.

To specify the default event handler, set this flag to `true`.

Otherwise, set this flag to `false` to specify a custom event handler.

Data Types: `logical`

`timeOut` — Timeout value for event handler
`500` milliseconds (default) | nonnegative integer

Timeout value for event handler for the Bloomberg EMSX service, specified as the comma-separated pair consisting of `'timeOut'` and a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types: `double`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

When the name-value pair argument `'useDefaultEventHandler'` or the same field for the structure `options` is set to `false`, `events` is an empty double.

`subs` — Bloomberg EMSX subscription list
subscription list object

Bloomberg EMSX subscription list, returned as a Bloomberg EMSX subscription list object.

Tips

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using this code.

```
t = timer('TimerFcn',{@c.eventhandler},'Period',1,...
          'ExecutionMode','fixedRate')
```

`t` is the MATLAB timer object. For details, see `timer`.

Version History

Introduced in R2013a

See Also

`emsx` | `getAllFieldMetaData` | `createOrderAndRoute` | `deleteRoute` | `modifyRoute` | `routeOrder` | `createOrderAndRouteWithStrat` | `orders` | `modifyOrder` | `deleteOrder` | `createOrder` | `close` | `timer` | `start` | `stop` | `delete`

Topics

“Create and Manage Bloomberg EMSX Order” on page 4-4
“Create and Manage Bloomberg EMSX Route” on page 4-8
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Workflow for Bloomberg EMSX” on page 4-16
“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

EMSX API Programmers Guide

createBasket

Create basket of Bloomberg EMSX orders

Syntax

```
events = createBasket(c,basket,order)
events = createBasket(c,basket,'timeOut',timeout)
createBasket( ___ ,'useDefaultEventHandler',false)
 ___  = createBasket(c,basket,options)
```

Description

`events = createBasket(c,basket,order)` creates a basket of Bloomberg EMSX orders using the Bloomberg EMSX connection, basket name, and order request. `createBasket` returns the order sequence numbers and status message using the default event handler.

`events = createBasket(c,basket,'timeOut',timeout)` specifies a timeout value for the execution of the default event handler.

`createBasket( ___ ,'useDefaultEventHandler',false)` creates a basket of Bloomberg EMSX orders using any of the previous input argument combinations and a custom event handler function. Write a custom event handler to process the events associated with creating a basket of orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue.

`___  = createBasket(c,basket,options)` uses the `options` structure to customize the output, which is useful for configuring and saving your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the `useDefaultEventHandler` field is set to `true`, and omit this output argument when the `useDefaultEventHandler` field is set to `false`.

Examples

Create Basket of Bloomberg EMSX Orders Using Default Event Handler

Using a Bloomberg EMSX connection, create a basket of Bloomberg EMSX orders.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
order1.EMSX_ORDER_TYPE = 'MKT';
```

```
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Create a basket of the two existing orders. Specify the basket name. Then, specify the order numbers in the `orders` structure.

```
basket = 'OrderBasket';
orders.EMSX_SEQUENCE = [int32(354646);int32(354777)];
events = createBasket(c,basket,orders)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: [2×1 double]
    MESSAGE: 'Orders added to Basket'
```

The default event handler processes the events associated with creating a basket of orders. `events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order numbers
- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Create Basket of Bloomberg EMSX Orders Using Timeout Value

Using a Bloomberg EMSX connection, create a basket of Bloomberg EMSX orders. Specify a timeout value.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
order1.EMSX_ORDER_TYPE = 'MKT';
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Create a basket of the two existing orders. Specify the basket name. Then, specify the order numbers as the structure `orders`. Specify the timeout value of 200 milliseconds by using the `'timeOut'` flag.

```
basket = 'OrderBasket';
orders.EMSX_SEQUENCE = [int32(354646);int32(354777)];
events = createBasket(c,basket,orders,'timeOut',200)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: [2×1 double]
    MESSAGE: 'Orders added to Basket'
```

`events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order numbers
- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Create Basket of Bloomberg EMSX Orders Using Custom Event Handler

Using a Bloomberg EMSX connection, create a basket of Bloomberg EMSX orders. Specify using a custom event handler function to process the events.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
order1.EMSX_ORDER_TYPE = 'MKT';
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Create a basket of the two existing orders. Specify the basket name. Then, specify the order numbers as the structure `orders`. Use a custom event handler function to process the events. You can use the sample event handler function `processEvent` or write your own custom event handler function. For this example, use `processEvent` to process the events.

```
basket = 'OrderBasket';
orders.EMSX_SEQUENCE = [int32(354646);int32(354777)];
createBasket(c,basket,orders,'useDefaultEventHandler',false)
processEvent(c)
```

```
CreateBasket = {

    EMSX_SEQUENCE[] = {

        354646, 354777

    }

    MESSAGE = 'Orders added to Basket'

}
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Create Basket of Bloomberg EMSX Orders Using Options Structure

Using a Bloomberg EMSX connection, create a basket of Bloomberg EMSX orders. Specify an additional option for a timeout value by using a structure.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
order1.EMSX_ORDER_TYPE = 'MKT';
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Create a basket of the two existing orders. Specify the basket name. Then, specify the order numbers as the structure `orders`. Specify an additional option for a timeout value of 200 milliseconds by using the `options` structure.

```
basket = 'OrderBasket';
orders.EMSX_SEQUENCE = [int32(354646);int32(354777)];
options.timeOut = 200;
events = createBasket(c,basket,orders,options)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: [2×1 double]
    MESSAGE: 'Orders added to Basket'
```

`events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order numbers
- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

c — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`basket` — Basket name
character vector | string scalar

Basket name, specified as a character vector or string scalar.

Example: `"OrderBasket"`

Data Types: `char` | `string`

`order` — Order request
structure

Order request, specified as a structure that contains the `EMSX_SEQUENCE` field. This field contains the order numbers. Convert the order numbers to a 32-bit signed integer by using `int32`.

Example: `int32(123456)`

Data Types: `struct`

`timeout` — Timeout value
`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value
structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents
double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2019b

See Also

`emsx` | `orders` | `routes` | `createOrder` | `routeOrder` | `modifyOrder` | `deleteOrder` | `groupRouteOrder` | `processEvent` | `close`

Topics

“Workflow for Bloomberg EMSX” on page 4-16
“Create and Manage Bloomberg EMSX Order” on page 4-4
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Writing and Running Custom Event Handler Functions” on page 1-26

groupRouteOrder

Route group of Bloomberg EMSX orders

Syntax

```
events = groupRouteOrder(c,order)
events = groupRouteOrder(c,order,'timeOut',timeout)
groupRouteOrder( ___ ,'useDefaultEventHandler',false)
 ___  = groupRouteOrder(c,order,options)
```

Description

`events = groupRouteOrder(c,order)` routes a group of Bloomberg EMSX orders using the Bloomberg EMSX connection and order request. `groupRouteOrder` returns the order sequence number and status message using the default event handler.

`events = groupRouteOrder(c,order,'timeOut',timeout)` specifies a timeout value for the execution of the default event handler.

`groupRouteOrder( ___ ,'useDefaultEventHandler',false)` routes a group of Bloomberg EMSX orders using any of the previous input argument combinations and a custom event handler function. Write a custom event handler to process the events associated with routing a group of orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue.

`___  = groupRouteOrder(c,order,options)` uses the `options` structure to customize the output, which is useful for configuring and saving your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the `useDefaultEventHandler` field is set to `true`, and omit this output argument when the `useDefaultEventHandler` field is set to `false`.

Examples

Route Group of Bloomberg EMSX Orders Using Default Event Handler

Using a Bloomberg EMSX connection, route a group of Bloomberg EMSX orders.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
order1.EMSX_ORDER_TYPE = 'MKT';
```

```
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Route the two existing orders. Specify the order numbers, broker, and hand instruction in the `order` structure.

```
order.EMSX_SEQUENCE{1} = int32(354646);
order.EMSX_SEQUENCE{2} = int32(354777);
order.EMSX_BROKER = 'BB';
order.EMSX_HAND_INSTRUCTION = 'ANY';
events = groupRouteOrder(c,order)
```

```
events =

  struct with fields:
```

```
    EMSX_SEQUENCE: 354646
    EMSX_ROUTE_ID: 1
    MESSAGE: 'Order Routed'
```

The default event handler processes the events associated with routing a group of orders. `events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order numbers
- `EMSX_ROUTE_ID` — Bloomberg EMSX route identifier
- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Group of Bloomberg EMSX Orders Using Timeout Value

Using a Bloomberg EMSX connection, route a group of Bloomberg EMSX orders. Specify a timeout value.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
order1.EMSX_ORDER_TYPE = 'MKT';
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Route the two existing orders. Specify the order numbers, broker, and hand instruction in the `order` structure. Specify an additional option for a timeout value of 200 milliseconds by using the `'timeOut'` flag.

```
order.EMSX_SEQUENCE{1} = int32(354646);
order.EMSX_SEQUENCE{2} = int32(354777);
order.EMSX_BROKER = 'BB';
order.EMSX_HAND_INSTRUCTION = 'ANY';
events = groupRouteOrder(c,order,'timeOut',200)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    EMSX_ROUTE_ID: 1
    MESSAGE: 'Order Routed'
```

`events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order numbers
- `EMSX_ROUTE_ID` — Bloomberg EMSX route identifier
- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Group of Bloomberg EMSX Orders Using Custom Event Handler Function

Using a Bloomberg EMSX connection, route a group of Bloomberg EMSX orders. Specify using a custom event handler function to process the events.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
order1.EMSX_ORDER_TYPE = 'MKT';
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Route the two existing orders. Specify the order numbers, broker, and hand instruction in the `order` structure. Use a custom event handler function to process the events. You can use the sample event handler function `processEvent` or write your own custom event handler function. For this example, use `processEvent` to process the events.

```
order.EMSX_SEQUENCE{1} = int32(354646);
order.EMSX_SEQUENCE{2} = int32(354777);
order.EMSX_BROKER = 'BB';
order.EMSX_HAND_INSTRUCTION = 'ANY';
groupRouteOrder(c,order,'useDefaultEventHandler',false)
processEvent(c)
```

```
Route = {

     EMSX_SEQUENCE = 354646

     EMSX_ROUTE_ID = 1

     MESSAGE = 'Order Routed'

     }
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Route Group of Bloomberg EMSX Orders Using Options Structure

Using a Bloomberg EMSX connection, route a group of Bloomberg EMSX orders. Specify an additional option for a timeout value by using a structure.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order1` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order1.EMSX_TICKER = 'IBM';
order1.EMSX_AMOUNT = int32(100);
order1.EMSX_ORDER_TYPE = 'MKT';
order1.EMSX_BROKER = 'BB';
order1.EMSX_TIF = 'DAY';
order1.EMSX_HAND_INSTRUCTION = 'ANY';
order1.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order1`.

```
events = createOrder(c,order1)
```

```
events =

  struct with fields:
```

```
    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Create another order request structure `order2` to define the order parameters. In this case, the code creates a buy market order for 200 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order2.EMSX_TICKER = 'IBM';
order2.EMSX_AMOUNT = int32(200);
order2.EMSX_ORDER_TYPE = 'MKT';
order2.EMSX_BROKER = 'BB';
order2.EMSX_TIF = 'DAY';
order2.EMSX_HAND_INSTRUCTION = 'ANY';
order2.EMSX_SIDE = 'BUY';
```

Create the second order using the Bloomberg EMSX connection `c` and `order2`.

```
events = createOrder(c,order2)

events =

  struct with fields:

    EMSX_SEQUENCE: 354777
    MESSAGE: 'Order created'
```

Route the two existing orders. Specify the order numbers, broker, and hand instruction in the `order` structure. Specify an additional option for a timeout value of 200 milliseconds by using the `options` structure.

```
order.EMSX_SEQUENCE{1} = int32(354646);
order.EMSX_SEQUENCE{2} = int32(354777);
order.EMSX_BROKER = 'BB';
order.EMSX_HAND_INSTRUCTION = 'ANY';
options.timeOut = 200;
events = groupRouteOrder(c,order,options)

events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    EMSX_ROUTE_ID: 1
    MESSAGE: 'Order Routed'
```

`events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order numbers
- `EMSX_ROUTE_ID` — Bloomberg EMSX route identifier

- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`order` — Order request

structure

Order request, specified as a structure that contains these fields:

- `EMSX_SEQUENCE` — Order numbers
- `EMSX_BROKER` — Broker
- `EMSX_HAND_INSTRUCTION` — Hand instruction

Convert the order numbers to a 32-bit signed integer by using `int32`.

Data Types: `struct`

`timeout` — Timeout value

`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value

structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2019b

See Also

`emsx` | `orders` | `routes` | `createOrder` | `routeOrder` | `modifyOrder` | `deleteOrder` | `createBasket` | `processEvent` | `close`

Topics

“Workflow for Bloomberg EMSX” on page 4-16
“Create and Manage Bloomberg EMSX Order” on page 4-4
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Writing and Running Custom Event Handler Functions” on page 1-26

manualFill

Fill Bloomberg EMSX orders manually

Syntax

```
events = manualFill(c,order)
events = manualFill(c,order,'timeOut',timeout)
manualFill( ___ ,'useDefaultEventHandler',false)
 ___  = manualFill(c,order,options)
```

Description

`events = manualFill(c,order)` manually fills a Bloomberg EMSX order using the Bloomberg EMSX connection and order request. `manualFill` returns the order sequence number and status message using the default event handler.

`events = manualFill(c,order,'timeOut',timeout)` specifies a timeout value for the execution of the default event handler.

`manualFill( ___ ,'useDefaultEventHandler',false)` manually fills a Bloomberg EMSX order using any of the previous input argument combinations and a custom event handler function. Write a custom event handler to process the events associated with manually filling an order. This syntax does not have an output argument because the custom event handler processes the contents of the event queue.

`___  = manualFill(c,order,options)` uses the `options` structure to customize the output, which is useful for configuring and saving your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the `useDefaultEventHandler` field is set to `true`, and omit this output argument when the `useDefaultEventHandler` field is set to `false`.

Examples

Manually Fill Bloomberg EMSX Order Using Default Event Handler

Using a Bloomberg EMSX connection, manually fill a Bloomberg EMSX order.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
```

```
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order`.

```
events = createOrder(c,order)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Manually fill the Bloomberg order. Specify the `manualorder` structure with the order number in the `events` structure.

```
manualorder.EMSX_SEQUENCE = int32(events.EMSX_SEQUENCE);
events = manualFill(c,manualorder)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order Filled'
```

The default event handler processes the events associated with manually filling the order. `events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Manually Fill Bloomberg EMSX Order Using Timeout Value

Using a Bloomberg EMSX connection, manually fill a Bloomberg EMSX order. Specify a timeout value.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force

set to DAY and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order`.

```
events = createOrder(c,order)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Manually fill the Bloomberg order. Specify the `manualorder` structure with the order number in the `events` structure. Specify the timeout value of 200 milliseconds by using the `'timeOut'` flag.

```
manualorder.EMSX_SEQUENCE = int32(events.EMSX_SEQUENCE);
events = manualFill(c,manualorder,'timeOut',200)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order Filled'
```

`events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Manually Fill Bloomberg EMSX Order Using Custom Event Handler Function

Using a Bloomberg EMSX connection, manually fill a Bloomberg EMSX order. Specify using a custom event handler function to process the events.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order`.

```
events = createOrder(c,order)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Manually fill the Bloomberg order. Specify the `manualorder` structure with the order number in the `events` structure. Use a custom event handler function to process the events. You can use the sample event handler function `processEvent` or write your own custom event handler function. For this example, use `processEvent` to process the events.

```
manualorder.EMSX_SEQUENCE = int32(events.EMSX_SEQUENCE);
manualFill(c,manualorder,'useDefaultEventHandler',false)
processEvent(c)
```

```
ManualFill = {

    EMSX_SEQUENCE = 354646

    MESSAGE = 'Order Filled'

}
```

Close the Bloomberg EMSX connection.

```
close(c)
```

Manually Fill Bloomberg EMSX Order Using Options Structure

Using a Bloomberg EMSX connection, manually fill a Bloomberg EMSX order. Specify an additional option for a timeout value by using a structure.

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage Bloomberg EMSX Order” on page 4-4.

Create the order request structure `order` to define the order parameters. In this case, the code creates a buy market order for 100 shares of IBM. The code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(100);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create the order using the Bloomberg EMSX connection `c` and `order`.

```
events = createOrder(c,order)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order created'
```

The default event handler processes the events associated with creating the order. `createOrder` returns `events` as a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number
- `MESSAGE` — Bloomberg EMSX message

Manually fill the Bloomberg order. Specify the `manualorder` structure with the order number in the `events` structure. Then, specify an additional option for a timeout value of 200 milliseconds by using the `options` structure.

```
manualorder.EMSX_SEQUENCE = int32(events.EMSX_SEQUENCE);
options.timeOut = 200;
events = manualFill(c,manualorder,options)
```

```
events =

  struct with fields:

    EMSX_SEQUENCE: 354646
    MESSAGE: 'Order Filled'
```

`events` is a structure that contains these fields:

- `EMSX_SEQUENCE` — Bloomberg EMSX order number

- `MESSAGE` — Bloomberg EMSX message

Close the Bloomberg EMSX connection.

```
close(c)
```

Input Arguments

`c` — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

`order` — Order request

structure

Order request, specified as a structure that contains the `EMSX_SEQUENCE` field. This field contains the order numbers. Convert the order numbers to a 32-bit signed integer by using `int32`.

Example: `int32(123456)`

Data Types: `struct`

`timeout` — Timeout value

`500` milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer. This integer denotes the time, in milliseconds, that the event handler listens to the event queue for each iteration of the code. The event handler can be a default or custom event handler.

Data Types: `double`

`options` — Options for custom event handler or timeout value

structure

Options for a custom event handler or timeout value, specified as a structure. To reuse the settings for specifying a custom event handler or timeout value for the event handler, use the `options` structure.

For example, specify using a custom event handler and a timeout value of 200 milliseconds.

```
options.useDefaultEventHandler = false;
options.timeOut = 200;
```

Data Types: `struct`

Output Arguments

`events` — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

Version History

Introduced in R2019b

See Also

`emsx` | `orders` | `routes` | `createOrder` | `routeOrder` | `modifyOrder` | `deleteOrder` | `createBasket` | `groupRouteOrder` | `processEvent` | `close`

Topics

“Workflow for Bloomberg EMSX” on page 4-16
“Create and Manage Bloomberg EMSX Order” on page 4-4
“Manage Bloomberg EMSX Order and Route” on page 4-12
“Writing and Running Custom Event Handler Functions” on page 1-26

cqg

Create CQG connection object

Description

The `cqg` function creates a `cqg` object, which represents a CQG connection. After you create a `cqg` object, you can use the object functions to create orders and retrieve historical, real-time, and intraday tick data.

Creation

Syntax

```
c = cqg
```

Description

`c = cqg` creates a CQG connection object `c`.

Properties

`Handle` — CQG handle

ActiveX® object

CQG handle, specified as an ActiveX object.

Example: `[1x1 COM.CQG_CQGCEL_4]`

`APIConfig` — API configuration type library specification

configuration object

API configuration type library specification, specified as a configuration object.

Example: `[1x1 Interface.CQG_4.0_Type_Library_-_Revised_API.ICQGAPIConfig]`

Object Functions

CQG Connection

startUp	Create CQG connection
shutDown	Close CQG connection
close	Close CQG connection

CQG Order Creation

createOrder	Create CQG order

CQG Data Retrieval

history	Request CQG historical data
realtime	Subscribe to CQG instrument
timeseries	Request CQG intraday tick data

Examples

Create CQG Connection Object

Create a CQG connection object.

```
c = cqg
```

```
c =

  cqg with properties:

      Handle: [1x1 COM.CQG_CQGCEL_4]
   APIConfig: [1x1 Interface.CQG_4.0_Type_Library_-_Revised_API.ICQGAPIConfig]
```

CQG connection object properties reflect the CQG ActiveX object `Handle` and the API configuration type library specification `APIConfig`.

Display the `Handle` property of `c`.

```
c.Handle
```

```
ans =

    COM.CQG_CQGCEL_4
```

After creating the `cqg` connection object, you can retrieve historical, real-time, and intraday tick data. For details, see `history`, `realtime`, and `timeseries`, respectively.

Close the CQG connection.

```
close(c)
```

Version History

Introduced in R2013b

See Also

Topics

“Data Server Connection Requirements” on page 1-3
“Workflow for CQG” on page 3-16
“Create CQG Orders” on page 3-20
“Request CQG Historical Data” on page 3-24
“Request CQG Intraday Tick Data” on page 3-27
“Request CQG Real-Time Data” on page 3-30

External Websites

CQG API Reference Guide

close

Close CQG connection

Syntax

```
close(c)
```

Description

`close(c)` closes CQG connection `c`.

Examples

Close the CQG Connection

Create the CQG connection object `c` using `cqg`.

```
c = cqg;
```

Create the CQG connection using `startUp`.

```
startUp(c)
```

Close the connection using the CQG connection object `c`.

```
close(c)
```

Input Arguments

c — CQG connection

connection object

CQG connection, specified as a CQG connection object created using `cqg`.

Version History

Introduced in R2013b

See Also

`cqg` | `shutDown`

Topics

“Create Order Using CQG” on page 3-18
“Create CQG Orders” on page 3-20
“Request CQG Historical Data” on page 3-24
“Request CQG Intraday Tick Data” on page 3-27
“Request CQG Real-Time Data” on page 3-30
“Workflow for CQG” on page 3-16

External Websites
CQG API Reference Guide

createOrder

Create CQG order

Syntax

```
o = createOrder(c,s,1,account,quantity)
o = createOrder(c,s,2,account,quantity,limitprice)
o = createOrder(c,s,3,account,quantity,stopprice)
o = createOrder(c,s,4,account,quantity,limitprice,stopprice)
```

Description

`o = createOrder(c,s,1,account,quantity)` creates a `CQGOrder` object `o` for a market order of `quantity` shares of CQG instrument `s` using the `CQGAccount` credentials object `account` over the CQG connection `c`.

`o = createOrder(c,s,2,account,quantity,limitprice)` creates a limit order using a CQG limit price `limitprice`.

`o = createOrder(c,s,3,account,quantity,stopprice)` creates a stop order using a CQG stop price `stopprice`.

`o = createOrder(c,s,4,account,quantity,limitprice,stopprice)` creates a stop limit order using CQG limit and stop prices, `limitprice` and `stopprice`.

Examples

Create and Place a Market Order Using a CQGInstrument Object

To create and place a market order for shares of an instrument with the CQG Trader Com API using a `CQGInstrument` object to specify the instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with the connection status. Set up the API configuration properties. Then, register event handlers for tracking events associated with the instrument subscription, order and account. Subscribe to the instrument and create the `CQGInstrument` object `cqgInst`. Then, set up the account credentials `accountHandle`. For an example demonstrating these activities, see “Create CQG Orders” on page 3-20. See *CQG API Reference Guide* to learn more about event handlers, API configuration properties, and `CQGInstrument` object.

Create a market order that buys one share of the subscribed security `cqgInst` using the account credentials `accountHandle`.

```
quantity = 1;

oMarket = createOrder(c,cqgInst,1,accountHandle,quantity);
oMarket.Place
```

```
ans =
    OrderChanged
```

The `CQGOrder` object `oMarket` contains the order. The CQG API executes the market order using the CQG API function `Place`. After execution, the order status changes.

Close the CQG connection.

```
shutDown(c)
```

Create and Place a Market Order Using a CQG Instrument Character Vector

To create and place a market order for shares of an instrument with the CQG Trader Com API, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register event handlers for tracking events associated with instrument subscription, order, and account. Subscribe to the instrument. Then, set up the account credentials `accountHandle`. For an example demonstrating these activities, see “Create CQG Orders” on page 3-20. To learn more about the event handlers and the API configuration properties, see the *CQG API Reference Guide*.

Create a market order that buys one share of the previously subscribed security `'EZC'` using the defined account credentials `accountHandle`.

```
cqgInstrumentName = 'EZC';
quantity = 1;

oMarket = createOrder(c,cqgInstrumentName,1,accountHandle, ...
    quantity);
oMarket.Place
```

```
ans =
    OrderChanged
```

The `CQGOrder` object `oMarket` contains the order. The CQG API executes the market order using the CQG API function `Place`. After execution, the order status changes.

Close the CQG connection.

```
shutDown(c)
```

Create and Place a Limit Order

To create and place a limit order for shares of an instrument with the CQG Trader Com API using a `CQGInstrument` object to specify the instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register event handlers for tracking events associated with instrument subscription, order and account. Subscribe to the instrument and create the `CQGInstrument` object `cqgInst`. Then, set up the account credentials `accountHandle`. For an example demonstrating these activities, see “Create CQG Orders” on page 3-20. See *CQG API Reference Guide* to learn more about the event handlers, the API configuration properties, and the `CQGInstrument` object.

To create a limit order, you can use the bid price. Extract the CQG bid object `qtBid` from the previously defined `CQGInstrument` object `cqgInst`.

```
qtBid = cqgInst.get('Bid');
```

Create a limit order that buys one share of the previously subscribed security `cqgInst` using the previously defined account credentials `accountHandle` and `qtBid` for the limit price.

```
quantity = 1;
limitprice = qtBid.get('Price');

oLimit = createOrder(c,cqgInst,2,accountHandle,quantity, ...
    limitprice);
oLimit.Place

ans =
    OrderChanged
```

The `CQGOrder` object `oLimit` contains the order. The CQG API executes the limit order using the CQG API function `Place`. After execution, the order status changes.

Close the CQG connection.

```
shutDown(c)
```

Create and Place a Stop Order

To create and place a stop order for shares of an instrument with the CQG Trader Com API using a `CQGInstrument` object to specify the instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register event handlers for tracking events associated with instrument subscription, order and account. Subscribe to the instrument and create the `CQGInstrument` object `cqgInst`. Then, set up the account credentials `accountHandle`. For an example demonstrating these activities, see “Create CQG Orders” on page 3-20. See *CQG API Reference Guide* to learn more about the event handlers, the API configuration properties, and the `CQGInstrument` object.

To create a stop order, you can use the trade price. Extract the CQG trade object `qtTrade` from the previously defined `CQGInstrument` object `cqgInst`.

```
qtTrade = cqgInst.get('Trade');
```

Create a stop order that buys one share of the previously subscribed security `cqgInst` using the previously defined account credentials `accountHandle` and `qtTrade` for the stop price.

```
quantity = 1;
stopprice = qtTrade.get('Price');

oStop = createOrder(c,cqgInst,3,accountHandle,quantity, ...
    stopprice);
oStop.Place

ans =
    OrderChanged
```

The `CQGOrder` object `oStop` contains the order. The CQG API executes the stop order using the CQG API function `Place`. After execution, the order status changes.

Close the CQG connection.

```
shutDown(c)
```

Create and Place a Stop Limit Order

To create and place a stop limit order for shares of an instrument with the CQG Trader Com API using a `CQGInstrument` object to specify the instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register event handlers for tracking events associated with instrument subscription, order and account. Subscribe to the instrument and create the `CQGInstrument` object `cqgInst`. Then, set up the account credentials `accountHandle`. For an example demonstrating these activities, see “Create CQG Orders” on page 3-20. See *CQG API Reference Guide* to learn more about the event handlers, the API configuration properties, and the `CQGInstrument` object.

To create a stop limit order, you can use the bid and trade prices. Extract the CQG bid object `qtBid` and the CQG trade object `qtTrade` from the previously defined `CQGInstrument` object `cqgInst`.

```
qtBid = cqgInst.get('Bid');
qtTrade = cqgInst.get('Trade');
```

Create a stop limit order that buys one share of the subscribed security `cqgInst` using the defined account credentials `accountHandle` and `qtBid` for the limit price and `qtTrade` for the stop price.

```
quantity = 1;
limitprice = qtBid.get('Price');
stopprice = qtTrade.get('Price');

oStopLimit = createOrder(c,cqgInst,4,accountHandle,quantity, ...
    limitprice,stopprice);
oStopLimit.Place
```

```
ans =
    OrderChanged
```

The `CQGOrder` object `oStopLimit` contains the order. The CQG API executes the stop limit order using the CQG API function `Place`. After execution, the order status changes.

Close the CQG connection.

```
shutDown(c)
```

Input Arguments

c — CQG connection

connection object

CQG connection, specified as a CQG connection object created using `cqg`.

s — CQG instrument name

character vector | string scalar | `CQGInstrument` object

CQG instrument name, specified as a character vector, string scalar, or `CQGInstrument` object, denoting the instrument or security for the order transaction. For more information about creating a

`CQGInstrument` object, see the *CQG API Reference Guide*. For a list of CQG instrument names, see Tradable Symbols.

`account` — CQG account credentials
`CQGAccount` object

CQG account credentials, specified as a `CQGAccount` object. This object encapsulates all data pertinent to your account. For more information about creating a `CQGAccount` object, see *CQG API Reference Guide*.

`quantity` — CQG order quantity
numeric scalar

CQG order quantity, specified as a numeric scalar denoting the number of shares to order. A positive number denotes a buy and a negative number denotes a sell.

Data Types: `double`

`limitprice` — CQG limit price
double

CQG limit price, specified as a double denoting the limit order price.

Data Types: `double`

`stopprice` — CQG stop price
double

CQG stop price, specified as a double denoting the stop order price.

Data Types: `double`

Output Arguments

`o` — CQG order
`CQGOrder` object

CQG order, returned as a `CQGOrder` object. This object encapsulates all data necessary to execute a CQG order. For more information about creating a `CQGOrder` object, see *CQG API Reference Guide*.

Version History

Introduced in R2013b

See Also

`cqg` | `history` | `realtime` | `timeseries`

Topics
“Create Order Using CQG” on page 3-18
“Create CQG Orders” on page 3-20
“Request CQG Historical Data” on page 3-24
“Request CQG Intraday Tick Data” on page 3-27
“Request CQG Real-Time Data” on page 3-30
“Workflow for CQG” on page 3-16

External Websites
CQG API Reference Guide

history

Request CQG historical data

Syntax

```
history(c,s,startdate,enddate,period)
history(c,s,startdate,enddate,period,x)
```

Description

`history(c,s,startdate,enddate,period)` requests CQG historical data asynchronously with bar size `period` between `startdate` and `enddate` for CQG instrument name `s` with CQG connection `c`.

`history(c,s,startdate,enddate,period,x)` requests CQG historical data asynchronously with additional request properties `x`.

Examples

Request CQG Historical Data

To request daily historical data for an instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see “Request CQG Historical Data” on page 3-24. See *CQG API Reference Guide* to learn more about event handlers and the API configuration properties.

Request historical daily data for instrument `XYZ.XYZ` for the last 10 days. `XYZ.XYZ` is a sample instrument name. To request historical data for your instrument, substitute the symbol name in `instrument`.

```
instrument = {'Close(XYZ.XYZ)','Open(XYZ.XYZ)'};
startdate = floor(now) - 10;
enddate = floor(now);
period = 'hpDaily';

history(c,instrument,startdate,enddate,period)
pause(1)
```

MATLAB writes variable `cqgHistoryData` to the Workspace browser.

Display `cqgHistoryData`.

```
cqgHistoryData
```

```
cqgHistoryData =
   1.0e+05 *
    7.3533    0.0063    0.0063
    7.3533    0.0064    0.0064
```

```
    7.3533    0.0065    0.0065
    7.3534    0.0065    0.0065
    7.3534    0.0066    0.0066
    7.3534    0.0065    0.0065
    7.3534    0.0066    0.0066
    7.3534    0.0066    0.0066
    7.3534    0.0064    0.0064
```

Each row in `cqgHistoryData` represents data for 1 day. The columns in `cqgHistoryData` show the numerical representation of the timestamp, the close price, and the open price for the instrument during the day.

Close the CQG connection.

```
close(c)
```

Request CQG Historical Data with Additional Request Properties

To request daily historical data for an instrument with an additional property, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see “Request CQG Historical Data” on page 3-24. See *CQG API Reference Guide* to learn more about event handlers and the API configuration properties.

Pass an additional optional request property by creating the structure `x` and setting the optional property.

```
x.UpdatesEnabled = false;
```

For additional optional properties you can set, see *CQG API Reference Guide*.

Request historical daily data for instrument `XYZ.XYZ` for the last 10 days using the additional optional request property `x`. `XYZ.XYZ` is a sample instrument name. To request historical data for your instrument, substitute the symbol name in `instrument`.

```
instrument = {'Close(XYZ.XYZ)','Open(XYZ.XYZ)'};
startdate = floor(now) - 10;
enddate = floor(now);
period = 'hpDaily';

history(c,instrument,startdate,enddate,period,x)
pause(1)
```

MATLAB writes the variable `cqgHistoryData` to the Workspace browser.

Display `cqgHistoryData`.

```
cqgHistoryData
```

```
cqgHistoryData =
   1.0e+05 *
    7.3533    0.0063    0.0063
    7.3533    0.0064    0.0064
    7.3533    0.0065    0.0065
```

```
    7.3534    0.0065    0.0065
    7.3534    0.0066    0.0066
    7.3534    0.0065    0.0065
    7.3534    0.0066    0.0066
    7.3534    0.0066    0.0066
    7.3534    0.0064    0.0064
```

Each row in `cqgHistoryData` represents data for 1 day. The columns in `cqgHistoryData` show the numerical representation of the timestamp, the close price, and the open price for the instrument during the day.

Close the CQG connection.

```
close(c)
```

Input Arguments

`c` — CQG connection

connection object

CQG connection, specified as a CQG connection object created using `cqg`.

`s` — CQG instrument name

character vector | string scalar

CQG instrument name, specified as a character vector or string scalar that identifies the instrument or security. For a list of CQG instrument names, see Tradable Symbols.

Data Types: `char` | `string`

`startdate` — Start date

character vector | string scalar | numeric scalar

Start date, specified as a character vector, string scalar, or numeric scalar.

Data Types: `double` | `char` | `string`

`enddate` — End date

character vector | string scalar | numeric scalar

End date, specified as a character vector, string scalar, or numeric scalar.

Data Types: `double` | `char` | `string`

`period` — Bar size

`'hpDaily'` (default) | `'hpWeekly'` | `'hpMonthly'` | `'hpQuarterly'` | `'hpSemiannual'` | `'hpYearly'`

Bar size, specified as one of the above values predetermined by the CQG API that denotes the length of time to collect data.

`x` — CQG request properties

request properties structure

CQG request properties, specified as a CQG request properties structure. Create this structure by writing MATLAB code to set additional optional request properties. For additional optional properties you can set, see *CQG API Reference Guide*.

Example: `x.UpdatesEnabled = false;`

Data Types: `struct`

Version History

Introduced in R2013b

See Also

`cqg` | `createOrder` | `timeseries` | `realtime`

Topics

“Create CQG Orders” on page 3-20
“Request CQG Historical Data” on page 3-24
“Request CQG Intraday Tick Data” on page 3-27
“Request CQG Real-Time Data” on page 3-30
“Workflow for CQG” on page 3-16

External Websites

CQG API Reference Guide

realtime

Subscribe to CQG instrument

Syntax

```
realtime(c,s)
```

Description

`realtime(c,s)` subscribes to a CQG instrument `s` using CQG connection `c`.

Examples

Subscribe to the CQG Instrument

To subscribe to the CQG instrument and get current data, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with instrument subscription. For an example demonstrating these activities, see “Request CQG Real-Time Data” on page 3-30. See *CQG API Reference Guide* to learn more about event handlers and the API configuration properties.

With the connection established, subscribe to the instrument. The instrument name must be formatted in the CQG long symbol view. For example, to subscribe to a security tied to corn, type the following.

```
instrument = 'F.US.EZC';
realtime(c,instrument)
```

MATLAB writes the structure variable `cqgDataEZC` to the Workspace browser.

Display `cqgDataEZC`.

```
cqgDataEZC(1,1)
```

```
ans =
              Price: {15x1 cell}
             Volume: {15x1 cell}
    ServerTimestamp: {15x1 cell}
          Timestamp: {15x1 cell}
               Type: {15x1 cell}
               Name: {15x1 cell}
            IsValid: {15x1 cell}
         Instrument: {15x1 cell}
          HasVolume: {15x1 cell}
```

`cqgDataEZC` returns the current quotes for the security.

Display data in the `Price` property of `cqgDataEZC`.

```
cqgDataEZC(1,1).Price
```

```
ans =
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [   660.5000]
    []
    []
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [-2.1475e+09]
    [   660.5000]
    [-2.1475e+09]
```

Close the CQG connection.

```
close(c)
```

Input Arguments

c — CQG connection

connection object

CQG connection, specified as a CQG connection object created using `cqg`.

s — CQG instrument name

character vector | string scalar

CQG instrument name, specified as a character vector or string scalar that identifies the instrument or security. For a list of CQG instrument names, see Tradable Symbols.

Data Types: `char` | `string`

Version History

Introduced in R2013b

See Also

`cqg` | `createOrder` | `history` | `timeseries`

Topics

“Create Order Using CQG” on page 3-18
“Create CQG Orders” on page 3-20
“Request CQG Historical Data” on page 3-24
“Request CQG Intraday Tick Data” on page 3-27
“Request CQG Real-Time Data” on page 3-30
“Workflow for CQG” on page 3-16

External Websites

CQG API Reference Guide

shutDown

Close CQG connection

Syntax

```
shutDown(c)
```

Description

`shutDown(c)` closes the CQG connection `c`.

Examples

Close the CQG Connection

Create the CQG connection object using `cqg`.

```
c = cqg;
```

Create the CQG connection using `startUp`.

```
startUp(c)
```

Close the CQG connection.

```
shutDown(c)
```

Alternatively, close the CQG connection using `close`.

```
close(c)
```

Input Arguments

c — CQG connection

connection object

CQG connection, specified as a CQG connection object created using `cqg`.

Version History

Introduced in R2013b

See Also

`startUp` | `cqg` | `close`

Topics

“Create CQG Orders” on page 3-20
“Request CQG Historical Data” on page 3-24

“Request CQG Intraday Tick Data” on page 3-27
“Request CQG Real-Time Data” on page 3-30
“Workflow for CQG” on page 3-16

External Websites
CQG API Reference Guide

startUp

Create CQG connection

Syntax

```
startUp(c)
```

Description

`startUp(c)` creates the CQG connection `c`.

Examples

Create the CQG Connection

Create the CQG connection object using `cqg`.

```
c = cqg;
```

Create the CQG connection.

```
startUp(c)
```

Close the CQG connection.

```
close(c)
```

Input Arguments

c — CQG connection

connection object

CQG connection, specified as a CQG connection object created using `cqg`.

Version History

Introduced in R2013b

See Also

`shutDown` | `cqg` | `close`

Topics

“Create Order Using CQG” on page 3-18
“Create CQG Orders” on page 3-20
“Request CQG Historical Data” on page 3-24
“Request CQG Intraday Tick Data” on page 3-27
“Request CQG Real-Time Data” on page 3-30
“Workflow for CQG” on page 3-16

External Websites
CQG API Reference Guide

timeseries

Request CQG intraday tick data

Syntax

```
timeseries(c,s,startdate,enddate)
timeseries(c,s,startdate,enddate,[],x)

timeseries(c,s,startdate,enddate,intraday)
timeseries(c,s,startdate,enddate,intraday,x)
```

Description

`timeseries(c,s,startdate,enddate)` requests CQG raw intraday tick data asynchronously between `startdate` and `enddate` for CQG instrument name `s` with CQG connection `c`.

`timeseries(c,s,startdate,enddate,[],x)` requests CQG raw intraday tick data asynchronously without timed bar data using additional request properties `x`.

`timeseries(c,s,startdate,enddate,intraday)` requests CQG timed bar data asynchronously with the aggregated bar value `intraday`.

`timeseries(c,s,startdate,enddate,intraday,x)` requests CQG timed bar data asynchronously with additional request properties `x`.

Examples

Request CQG Intraday Tick Data

To request intraday tick data for an instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see “Request CQG Intraday Tick Data” on page 3-27. See *CQG API Reference Guide* to learn more about event handlers and the API configuration properties.

Request intraday tick data for instrument `XYZ.XYZ` for the last 2 days. `XYZ.XYZ` is a sample instrument name. To request intraday tick data for your instrument, substitute the symbol name in `instrument`.

```
instrument = 'XYZ.XYZ';
startdate = now - 2;
enddate = now;

timeseries(c,instrument,startdate,enddate)
```

MATLAB writes the structure variable `cqgTickData` to the Workspace browser.

Display `cqgTickData`.

```
cqgTickData
```

```
cqgTickData =
              Timestamp: {2x1 cell}
                  Price: [2x1 double]
                 Volume: [2x1 double]
              PriceType: {2x1 cell}
         CorrectionType: {2x1 cell}
    SalesConditionLabel: {2x1 cell}
     SalesConditionCode: [2x1 double]
          ContributorId: {2x1 cell}
      ContributorIdCode: [2x1 double]
            MarketState: {2x1 cell}
```

`cqgTickData` returns intraday tick data for the specified instrument.

Display the data in the `Timestamp` property of `cqgTickData`.

```
cqgTickData.Timestamp
```

```
ans =
    '4/17/2013 2:14:00 PM'
    '4/18/2013 2:14:00 PM'
```

Close the CQG connection.

```
close(c)
```

Request CQG Intraday Tick Data with Additional Properties

To request intraday tick data for an instrument with an additional property, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see “Request CQG Intraday Tick Data” on page 3-27. See *CQG API Reference Guide* to learn more about event handlers and the API configuration properties.

Pass an additional optional request property by creating the structure `x`, and setting the optional property. To see only bid tick data, for example, set `TickFilter` to `'tfBid'`.

```
x.TickFilter = 'tfBid';
```

`TickFilter` and `SessionsFilter` are the only valid additional optional properties for calling `timeseries` without a timed bar request. For additional property values you can set, see *CQG API Reference Guide*.

Request intraday tick data for instrument `XYZ.XYZ` for the last 2 days using the additional optional request property `x`. `XYZ.XYZ` is a sample instrument name. To request intraday tick data for your instrument, substitute the symbol name in `instrument`.

```
instrument = 'XYZ.XYZ';
startdate = now - 2;
enddate = now;

timeseries(c,instrument,startdate,enddate,[],x)
```

MATLAB writes the variable `cqgTickData` to the Workspace browser.

Display `cqgTickData`.

```
cqgTickData
```

```
cqgTickData =
              Timestamp: {2x1 cell}
                  Price: [2x1 double]
                 Volume: [2x1 double]
              PriceType: {2x1 cell}
         CorrectionType: {2x1 cell}
    SalesConditionLabel: {2x1 cell}
     SalesConditionCode: [2x1 double]
          ContributorId: {2x1 cell}
      ContributorIdCode: [2x1 double]
            MarketState: {2x1 cell}
```

`cqgTickData` returns intraday tick data for the specified instrument.

Display the data in the `Timestamp` property of `cqgTickData`.

```
cqgTickData.Timestamp
```

```
ans =
    '4/17/2013 2:14:00 PM'
    '4/18/2013 2:14:00 PM'
```

Close the CQG connection.

```
close(c)
```

Request CQG Timed Bar Data

To request timed bar data for an instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see “Request CQG Intraday Tick Data” on page 3-27. See *CQG API Reference Guide* to learn more about event handlers and the API configuration properties.

Request timed bar data for instrument `XYZ.XYZ` for the last fraction of a day. `XYZ.XYZ` is a sample instrument name. To request timed bar data for your instrument, substitute the symbol name in `instrument`.

```
instrument = 'XYZ.XYZ';
startdate = now - .1;
enddate = now;
intraday = 1;

timeseries(c,instrument,startdate,enddate,intraday)
```

MATLAB writes variable `cqgTimedBarData` to the Workspace browser.

Display `cqgTimedBarData`.

```
cqgTimedBarData
```

```
cqgTimedBarData =
   1.0e+09 *
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    ...
```

`cqgTimedBarData` returns timed bar data for the specified instrument. The columns of `cqgTimedBarData` display data corresponding to the timestamp, open price, high price, low price, close price, mid-price, HLC3, average price, and tick volume.

Close the CQG connection.

```
close(c)
```

Request CQG Timed Bar Data with Additional Properties

To request timed bar data for an instrument with an additional property, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see “Request CQG Intraday Tick Data” on page 3-27. See *CQG API Reference Guide* to learn more about event handlers and the API configuration properties.

Pass an additional optional request property by creating the structure `x`, and setting the optional property.

```
x.UpdatesEnabled = false;
```

For additional optional properties you can set, see *CQG API Reference Guide*.

Request timed bar data for instrument `XYZ.XYZ` for the last fraction of a day using the additional optional request property `x`. `XYZ.XYZ` is a sample instrument name. To request timed bar data for your instrument, substitute the symbol name in `instrument`.

```
instrument = 'XYZ.XYZ';
startdate = now - .1;
enddate = now;
intraday = 1;

timeseries(c,instrument,startdate,enddate,intraday,x)
```

MATLAB writes the variable `cqgTimedBarData` to the Workspace browser.

Display `cqgTimedBarData`.

```
cqgTimedBarData
```

```
cqgTimedBarData =
   1.0e+09 *
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475
    ...
```

`cqgTimedBarData` returns timed bar data for the specified instrument. The columns of `cqgTimedBarData` display data corresponding to the timestamp, open price, high price, low price, close price, mid-price, HLC3, average price, and tick volume.

Close the CQG connection.

```
close(c)
```

Input Arguments

`c` — CQG connection

connection object

CQG connection, specified as a CQG connection object created using `cqg`.

`s` — CQG instrument name

character vector | string scalar

CQG instrument name, specified as a character vector or string scalar that identifies the instrument or security. For a list of CQG instrument names, see Tradable Symbols.

Data Types: `char` | `string`

`startdate` — Start date

character vector | string scalar | numeric scalar

Start date, specified as a character vector, string scalar, or numeric scalar.

Data Types: `double` | `char` | `string`

`enddate` — End date

character vector | string scalar | numeric scalar

End date, specified as a character vector, string scalar, or numeric scalar.

Data Types: `double` | `char` | `string`

`intraday` — Aggregated bar value

numeric scalar | `[]`

Aggregated bar value, specified as a numeric scalar from 1.0 to 1440.0. If you want to call `timeseries` to return intraday tick data with additional properties without timed bar data, then enter `[]` for this argument.

Data Types: `double`

`x` — CQG request properties

request properties structure

CQG request properties, specified as a CQG request properties structure. Create this structure by writing MATLAB code to set additional optional request properties. For additional optional properties you can set, see *CQG API Reference Guide*.

Example: `x.UpdatesEnabled = false;`

Data Types: `struct`

Version History

Introduced in R2013b

See Also

`cqg` | `createOrder` | `history` | `realtime`

Topics

“Create CQG Orders” on page 3-20
“Request CQG Historical Data” on page 3-24
“Request CQG Intraday Tick Data” on page 3-27
“Request CQG Real-Time Data” on page 3-30
“Workflow for CQG” on page 3-16

External Websites

CQG API Reference Guide

ihsmarkitrs

IHS Markit connection

Description

The `ihsmarkitrs` function creates an `ihsmarkitrs` object, which represents an IHS Markit connection. First, you must obtain credentials from IHS Markit. For credentials, see the IHS Markit website. After you create an `ihsmarkitrs` object, you can use the object functions to retrieve factor, security, signal, and universe information.

Creation

Syntax

```
c = ihsmarkitrs(username,password)
```

Description

`c = ihsmarkitrs(username,password)` creates an IHS Markit connection using a password and sets the Username property.

Input Arguments

`password` — Password
character vector | string scalar

Password, specified as a character vector or string scalar. For credentials, see the IHS Markit website.

Data Types: `char` | `string`

Properties

`Username` — User name
character vector | string scalar

User name, specified as a character vector or string scalar. For credentials, see the IHS Markit website.

Data Types: `char` | `string`

`TimeOut` — Timeout
`100` (default) | numeric scalar

Timeout, specified as a numeric scalar that indicates the number of seconds to wait for data to return before canceling the request.

Example: `10`

Data Types: `double`

Object Functions

factorgroups	Retrieve factor group information
factors	Retrieve factor information
security	Retrieve security information
signals	Retrieve signal information
universes	Retrieve universe information

Examples

Retrieve Factor Groups

Using an IHS Markit connection, retrieve factor groups.

Create an IHS Markit connection using your user name and password.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password)
```

```
c =

  ihsmarkitrs with properties:

    Username: 'ABCDEF'
     TimeOut: 100
```

`c` is an `ihsmarkitrs` object with the `Username` and `TimeOut` properties. The appearance of the `ihsmarkitrs` object indicates a successful connection. The `Username` property contains your IHS Markit user name. The `TimeOut` property specifies waiting for a maximum of 100 seconds to return factor group data before canceling the request.

Retrieve factor group data using the IHS Markit connection. `d` is a table that contains factor group information.

```
d = factorgroups(c);
```

Display the first few rows of factor group data.

```
head(d)
```

```
ans =

  8×5 table

    id              name              hasCompositeModels    hasSubCompositeModels    hasBasicFac
    __    ____________________________    __________________    _____________________    ___________

     2    'Deep Value'                          true                  true                  true
     3    'Earnings Momentum'                   true                  true                  true
     4    'Earnings Quality'                    true                  true                  true
     5    'Historical Growth'                   true                  true                  true
     7    'Liquidity, Risk & Size'              true                  true                  true
     8    'Management Quality'                  true                  true                  true
     9    'Price Momentum'                      true                  true                  true
    10    'Relative Value'                      true                  true                  true
```

d contains these variables:

- Identification number of the factor group
- Name of the factor group
- Whether the factor group contains composite factors
- Whether the factor group contains subcomposite factors
- Whether the factor group contains basic factors

Version History

Introduced in R2018b

See Also

Topics

“Retrieve Factor Rank Data for Portfolio Selection” on page 13-2
“IHS Markit Error Messages” on page 13-4

External Websites

IHS Markit
IHS Markit Research Signals REST Documentation

factorgroups

Retrieve factor group information

Syntax

```
d = factorgroups(c)
```

Description

`d = factorgroups(c)` returns factor group information using the IHS Markit connection.

Examples

Retrieve Factor Groups

Using an IHS Markit connection, retrieve factor groups.

Create an IHS Markit connection using your user name and password.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password)
```

```
c =

  ihsmarkitrs with properties:

    Username: 'ABCDEF'
     TimeOut: 100
```

`c` is an `ihsmarkitrs` object with the `Username` and `TimeOut` properties. The appearance of the `ihsmarkitrs` object indicates a successful connection. The `Username` property contains your IHS Markit user name. The `TimeOut` property specifies waiting for a maximum of 100 seconds to return factor group data before canceling the request.

Retrieve factor group data using the IHS Markit connection. `d` is a table that contains factor group information.

```
d = factorgroups(c);
```

Display the first few rows of factor group data.

```
head(d)
```

```
ans =

  8×5 table

    id              name              hasCompositeModels    hasSubCompositeModels    hasBasicFac
    __    ____________________________    __________________    _____________________    ___________
```

```
     2    'Deep Value'                    true                    true                    true
     3    'Earnings Momentum'             true                    true                    true
     4    'Earnings Quality'              true                    true                    true
     5    'Historical Growth'             true                    true                    true
     7    'Liquidity, Risk & Size'        true                    true                    true
     8    'Management Quality'            true                    true                    true
     9    'Price Momentum'                true                    true                    true
    10    'Relative Value'                true                    true                    true
```

d contains these variables:

- Identification number of the factor group
- Name of the factor group
- Whether the factor group contains composite factors
- Whether the factor group contains subcomposite factors
- Whether the factor group contains basic factors

Input Arguments

c — IHS Markit connection

`ihsmarkitrs` object

IHS Markit connection, specified as an `ihsmarkitrs` object.

Output Arguments

d — Factor group information

table

Factor group information, specified as a table. The table contains information about available factor groups with these variables.

Variable	Description	Data Type
`id`	Identification number of the factor group	`double`
`name`	Name of the factor group	cell array of character vectors
`hasCompositeModels`	Whether the factor group contains composite factors	`logical`
`hasSubCompositeModels`	Whether the factor group contains subcomposite factors	`logical`
`hasBasicFactors`	Whether the factor group contains basic factors	`logical`

Version History

Introduced in R2018b

See Also

`ihsmarkitrs` | `factors` | `security` | `signals` | `universes`

Topics
“Retrieve Factor Rank Data for Portfolio Selection” on page 13-2
“IHS Markit Error Messages” on page 13-4

External Websites
IHS Markit
IHS Markit Research Signals REST Documentation

factors

Retrieve factor information

Syntax

```
d = factors(c)
d = factors(c,code)
d = factors(c,code,requesttype)
d = factors(c,code,'HistoryDetail',universeid)
```

Description

`d = factors(c)` returns a list of factors using an IHS Markit connection.

`d = factors(c,code)` returns factor information for the specified factor code or group name.

`d = factors(c,code,requesttype)` returns factor information based on the type of request.

`d = factors(c,code,'HistoryDetail',universeid)` returns the historical data for the specified factor within the specified universe using the `HistoryDetail` request.

Examples

Retrieve List of Factors

Using an IHS Markit connection, retrieve a list of factors.

Create an IHS Markit connection using your user name and password. `c` is an `ihsmarkitrs` object.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password);
```

Retrieve a list of factors using the IHS Markit connection. `d` is a table that contains the list of factors.

```
d = factors(c);
```

Display the first three variables to view the information for the first three factors.

```
d(1:3,1:3)
```

```
ans =

  3×3 table

    id        code                          name
   ____    __________    ____________________________________________

   7403    '3MChgGPA'    'Quarterly Change in Gross Profit to Assets'
   7404    '3MChgGPM'    'Quarterly Change in Gross Profit Margin'
      1    'ABR'         'Abnormal Return around QTR Earnings Release'
```

The variables are:

- `id` — Identification number of the factor
- `code` — Factor code
- `name` — Factor name

Retrieve Factor Information for Specific Factor Code

Using an IHS Markit connection, retrieve information for a specific factor by using the factor code.

Create an IHS Markit connection using your user name and password. `c` is an `ihsmarkitrs` object.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password);
```

Retrieve information for the factor with the code `ABR` using the IHS Markit connection. The `factors` function returns `d` as a table that contains the information for the specified factor.

```
code = 'ABR';
d = factors(c,code);
```

Display the first three variables of the table.

```
d(1,1:3)
```

```
ans =

  1×3 table

    id    code                          name
    __    _____    ____________________________________________

    1     'ABR'    'Abnormal Return around QTR Earnings Release'
```

The variables are:

- `id` — Identification number of the factor
- `code` — Factor code
- `name` — Factor name

Retrieve Factor Information for `Mapping` Request

Using an IHS Markit connection, retrieve information for a specific factor by using the factor code. Also, specify the `Mapping` request.

Create an IHS Markit connection using your user name and password. `c` is an `ihsmarkitrs` object.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password);
```

Retrieve information for the factor with the code `ABR` using the IHS Markit connection. Specify the `Mapping` type for the request. `d` is a table that has one variable, which lists the names of the universes that contain the specified factor.

```
code = 'ABR';
requesttype = 'Mapping';
d = factors(c,code,requesttype)
```

```
d =

  5×1 table

              universe
    ________________________________

    'QSG Bank Universe'
    'Markit US Large Cap'
    'Markit US Small Cap'
    'Markit US Total Cap'
    'US Total Cap Highly Shorted'
```

Retrieve Factor Information for `HistoryDetail` Request

Using an IHS Markit connection, retrieve information for a specific factor by using the factor code. Also, specify the `HistoryDetail` request.

Create an IHS Markit connection using your user name and password. `c` is an `ihsmarkitrs` object.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password);
```

Retrieve information for the factor with the code `ACI` using the IHS Markit connection. Specify the `HistoryDetail` type for the request and the `QSG World` universe. `d` is a table that contains the historical information for the specified factor.

```
code = 'ACI';
universeid = 'QSG World';
d = factors(c,code,'HistoryDetail',universeid)
```

```
d =

  4×8 table

    code     factorId    universeId      dataType       universe      freqType      startDate
    _____    ________    __________    _____________    __________    _________    ___________

    'ACI'       2           133        'Percentile'    'QSG World'    'Daily'      '10/01/2009'
    'ACI'       2           133        'Percentile'    'QSG World'    'Monthly'    '12/30/1988'
    'ACI'       2           133        'Rawratio'      'QSG World'    'Daily'      '10/01/2009'
    'ACI'       2           133        'Rawratio'      'QSG World'    'Monthly'    '12/30/1988'
```

`d` is a table with these variables:

- `code` — Factor code

- `factorId` — Identification number for the factor code
- `universeId` — Identification number for the universe
- `dataType` — Data reporting format
- `universe` — Universe name
- `freqType` — Frequency (periodicity)
- `startDate` — Start date of the factor in the universe
- `endDate` — End date of the factor in the universe

Input Arguments

c — IHS Markit connection

`ihsmarkitrs` object

IHS Markit connection, specified as an `ihsmarkitrs` object.

code — Factor code

character vector | string scalar

Factor code or group name, specified as a character vector or string scalar.

Example: `"ABR"`

Data Types: `char` | `string`

requesttype — Request type

character vector | string scalar

Request type, specified as the value `'ModelStructure'` or `'Mapping'`. Use the `'ModelStructure'` value to return a list of the composite factors that constitute the factor specified by the `code` input argument. Use the `'Mapping'` value to return a list of the names of universes that contain the specified factor.

You can specify each value as a character vector or string scalar.

universeid — Universe name

character vector | string scalar

Universe name, specified as a character vector or string scalar. Use `universeid` only with the `'HistoryDetail'` syntax.

Example: `'QSG World'`

Data Types: `char` | `string`

Output Arguments

d — Factor information

table

Factor information, returned as a table. The following table describes the variables in the returned data. Depending on the request type specified in the `requesttype` input argument or the `'HistoryDetail'` syntax, the returned table contains a subset of these variables.

Variable Name	Description	Data Type
`id`	Identification number of the factor	`double`
`code`	Factor code	cell array of character vectors
`name`	Factor name	cell array of character vectors
`description`	Factor description	cell array of character vectors
`dIntl`	Localized factor description	cell array of character vectors
`type`	Factor type	cell array of character vectors
`parentCode`	Factor parent code	cell array of character vectors
`rankingOrder`	Rank order (descending or ascending)	cell array of character vectors
`isRankAvailable`	Whether rank data is available for the factor	`logical`
`isZscoreAvailable`	Whether z-score data is available for the factor	`logical`
`isRawRatioAvailable`	Whether raw ratio data is available for the factor	`logical`
`modelType`	Model type of the factor	cell array of character vectors
`groupId`	Group identifier of the factor	`double`
`groupName`	Name of the factor group	cell array of character vectors
`factorId`	Factor identifier	`double`
`universe`	Universe name	cell array of character vectors
`universeId`	Identification number of the universe	`double`
`dataType`	Reporting format of the data	cell array of character vectors
`freqType`	Frequency (or periodicity) of the data	cell array of character vectors
`startDate`	Start date of the factor in the universe	cell array of character vectors
`endDate`	End date of the factor in the universe	cell array of character vectors
`childId`	Code of the factor within the composite factor	`double`
`childCode`	Name of the factor within the composite factor	cell array of character vectors
`weight`	Weight of the factor within the composite factor	`double`
`data`	Country code	cell array of character vectors

Version History

Introduced in R2018b

See Also

`ihsmarkitrs` | `factorgroups` | `security` | `signals` | `universes`

Topics

“Retrieve Factor Rank Data for Portfolio Selection” on page 13-2
“IHS Markit Error Messages” on page 13-4

External Websites

IHS Markit
IHS Markit Research Signals REST Documentation

security

Retrieve security information

Syntax

```
d = security(c,universeid,startdate,enddate)
d = security(c,universeid,startdate,enddate,identifier)
```

Description

`d = security(c,universeid,startdate,enddate)` returns security information using an IHS Markit connection, universe name, and date range.

`d = security(c,universeid,startdate,enddate,identifier)` specifies the type of security to retrieve.

Examples

Retrieve Security Information

Using an IHS Markit connection, retrieve security information using a date range within a specified universe.

Create an IHS Markit connection using your user name and password. `c` is an `ihsmarkitrs` object.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password);
```

Retrieve security information for the `US Total Cap` universe from January 1, 2017, through December 31, 2017, using the IHS Markit connection. `d` is a table that contains the security information.

```
universeid = "US Total Cap";
startdate = "2017-01-01";
enddate = "2017-12-31";
d = security(c,universeid,startdate,enddate);
```

Display the first few rows of security information.

```
head(d)
```

```
ans =

  8×7 table

      mid         startDate        endDate         cusip        sedol      ticker    quotCountry
    _________    ____________    ____________    _________    ________    ______    ___________

    1.3183e+05    '05/10/2013'    '01/01/2050'    '03265410'    '203206'    'ADI'         ''
    1.3262e+05    '05/10/2013'    '01/01/2050'    '00790310'    '200784'    'AMD'         ''
```

```
    1.3492e+05    '05/10/2013'    '01/01/2050'    '09676110'    '210775'    'BOBE'        ''
    1.4093e+05    '05/10/2013'    '01/01/2050'    '12550910'    '219647'    'CI'          ''
    1.4205e+05    '05/10/2013'    '01/01/2050'    '14428510'    '217750'    'CRS'         ''
    1.4224e+05    '05/10/2013'    '01/01/2050'    '12640810'    '216075'    'CSX'         ''
    1.4226e+05    '05/10/2013'    '01/01/2050'    '21683110'    '222260'    'CTB'         ''
    1.4344e+05    '05/10/2013'    '01/01/2050'    '24801910'    '226036'    'DLX'         ''
```

The variables are:

- `mid` — IHS Markit identification code
- `startDate` — Start date of the factor in the universe
- `endDate` — End date of the factor in the universe
- `cusip` — CUSIP security identifier
- `sedol` — SEDOL security identifier
- `ticker` — Ticker security identifier
- `quotCountry` — Market country of the security

Retrieve Security Information for Specific Security Type

Using an IHS Markit connection, retrieve security information using a date range within a specified universe. Specify the type of security to retrieve.

Create an IHS Markit connection using your user name and password. `c` is an `ihsmarkitrs` object.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password);
```

Retrieve security information for the `US Total Cap` universe from January 1, 2017, through December 31, 2017, using the IHS Markit connection. Specify retrieving only SEDOL security identifiers. `d` is a table that contains the security information.

```
universeid = "US Total Cap";
startdate = "2017-01-01";
enddate = "2017-12-31";
identifier = "sedol";
d = security(c,universeid,startdate,enddate,identifier);
```

Display the first few rows of security information.

```
head(d)
```

```
ans =

  8×5 table

      mid         startDate        endDate        sedol      quotCountry
    __________    ____________    ____________    ________    ___________

    1.3233e+05    '05/10/2013'    '01/01/2050'    '200111'        ''
      1.33e+05    '05/10/2013'    '01/01/2050'    '204617'        ''
    1.3353e+05    '05/10/2013'    '01/01/2050'    '206051'        ''
    1.3376e+05    '05/10/2013'    '01/01/2050'    '206650'        ''
```

```
    1.4304e+05    '05/10/2013'    '01/01/2050'    '227646'        ''
    1.4424e+05    '05/10/2013'    '01/01/2050'    '231380'        ''
    1.4498e+05    '05/10/2013'    '01/01/2050'    '232204'        ''
      1.46e+05    '05/10/2013'    '01/01/2050'    '234292'        ''
```

The variables are:

- `mid` — IHS Markit identification code
- `startDate` — Start date of the factor in the universe
- `endDate` — End date of the factor in the universe
- `sedol` — SEDOL security identifier
- `quotCountry` — Market country of the security

Input Arguments

`c` — IHS Markit connection

`ihsmarkitrs` object

IHS Markit connection, specified as an `ihsmarkitrs` object.

`universeid` — Universe name

character vector | string scalar

Universe name, specified as a character vector or string scalar.

Example: `'US Total Cap'`

Data Types: `char` | `string`

`startdate` — Start date

`datetime` array | numeric scalar | character vector | string scalar

Start date for a data request, specified as a `datetime` array, numeric scalar, character vector, or string scalar.

Example: `"2017-01-01"`

Data Types: `double` | `char` | `string` | `datetime`

`enddate` — End date

`datetime` array | numeric scalar | character vector | string scalar

End date for a data request, specified as a `datetime` array, numeric scalar, character vector, or string scalar.

Example: `"2017-12-31"`

Data Types: `double` | `char` | `string` | `datetime`

`identifier` — Security type

character vector | string scalar | cell array of character vectors | string array

Security type to retrieve, specified as one or more of these values: `'ticker'`, `'cusip'`, or `'sedol'`. You can specify these values as a character vector, string scalar, cell array of character vectors, or string array.

Output Arguments

d — Security information

table

Security information, returned as a table. The following table describes the variables in the returned data. (The variables for security type vary depending on the type that you specify in the `identifier` input argument.)

Variable Name	Description	Data Type
`mid`	IHS Markit identification code	`double`
`startDate`	Start date of the factor in the universe	cell array of character vectors
`endDate`	End date of the factor in the universe	cell array of character vectors
`cusip`	CUSIP security identifier	cell array of character vectors
`ticker`	Ticker security identifier	cell array of character vectors
`sedol`	SEDOL security identifier	cell array of character vectors
`quotCountry`	Market country of the security	cell array of character vectors

Version History

Introduced in R2018b

See Also

`ihsmarkitrs` | `factorgroups` | `factors` | `signals` | `universes`

Topics

“Retrieve Factor Rank Data for Portfolio Selection” on page 13-2
“IHS Markit Error Messages” on page 13-4

External Websites

IHS Markit
IHS Markit Research Signals REST Documentation

signals

Retrieve signal information

Syntax

```
d = signals(c,code,universeid,startdate,enddate)
d = signals(c,code,universeid,startdate,enddate,identifier)
d = signals(c,code,universeid,startdate,enddate,identifier,datatype)
d = signals(c,code,universeid,startdate,enddate,identifier,datatype,
monthlydata)
```

Description

`d = signals(c,code,universeid,startdate,enddate)` returns signal information using an IHS Markit connection, factor code, universe name, and date range.

`d = signals(c,code,universeid,startdate,enddate,identifier)` specifies the type of security to retrieve.

`d = signals(c,code,universeid,startdate,enddate,identifier,datatype)` specifies the data format for the returned signal information.

`d = signals(c,code,universeid,startdate,enddate,identifier,datatype, monthlydata)` specifies retrieval of monthly data.

Examples

Retrieve Signal Information

Using an IHS Markit connection, retrieve signal information using a factor and date range within a specified universe.

Create an IHS Markit connection using your user name and password. `c` is an `ihsmarkitrs` object.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password);
```

Retrieve signal information for the last 10 days using the IHS Markit connection. Specify the `ABR` factor code and `QSG World` universe. `ABR` is a sample factor code and `QSG World` is a sample universe. To retrieve signal information for your code and universe combination, substitute the factor code in `code` and universe in `universeid`. The `d` workspace variable is a table that contains signal information and the `date` and `data` variables.

```
code = 'ABR';
universeid = 'QSG World';
startdate = datetime('today')-10;
enddate = datetime('today');
d = signals(c,code,universeid,startdate,enddate);
```

Access the first few rows of signal information for the first day in the date range by using the `data` variable.

```
data = d.data{1};
head(data)
```

```
ans =

  8×2 table

    ticker    value
    ______    _____

    'VIRT'      1
    'SEDG'      1
    'CRTO'      1
    'BZUN'      1
    'FNGN'      1
    'CMG'       1
    'INGN'      1
    'ADAP'      1
```

The variables of the resulting table are `ticker` and `value`. The `ticker` variable contains the ticker security identifiers. The `value` variable contains the signal information for the corresponding security.

Retrieve Signal Information for SEDOL Security Type

Using an IHS Markit connection, retrieve signal information using a factor and date range within a specified universe. Specify the SEDOL security type.

Create an IHS Markit connection using your user name and password. `c` is an `ihsmarkitrs` object.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password);
```

Retrieve signal information for the last 10 days using the IHS Markit connection. Specify the `ABR` factor code and `QSG World` universe. `ABR` is a sample factor code and `QSG World` is a sample universe. To retrieve signal information for your code and universe combination, substitute the factor code in `code` and universe in `universeid`. Also, specify the SEDOL security type. `d` is a table that contains signal information and the `date` and `data` variables.

```
code = 'ABR';
universeid = 'QSG World';
startdate = datetime('today')-10;
enddate = datetime('today');
identifier = 'sedol';
d = signals(c,code,universeid,startdate,enddate,identifier);
```

Access the first few rows of signal information for the first day in the date range by using the `data` variable.

```
data = d.data{1};
head(data)
```

```
ans =

  8×2 table

     sedol      value
    ________    _____

    'BWTVWD'      1
    'BWC52Q'      1
    'BFPMB2'      1
    'BY2ZJ6'      1
    'B65V2X'      1
    'B0X7DZ'      1
    'BJSVLL'      1
    'BWY4XV'      1
```

The variables of the resulting table are `sedol` and `value`. The `sedol` variable contains the SEDOL security identifiers. The `value` variable contains the signal information for the corresponding security.

Retrieve Signal Information for Z-Score Data Format

Using an IHS Markit connection, retrieve signal information using a factor and date range within a specified universe. Specify the SEDOL security type and z-score data format.

Create an IHS Markit connection using your user name and password. `c` is an `ihsmarkitrs` object.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password);
```

Retrieve signal information for the last 10 days using the IHS Markit connection. Specify the `ABR` factor code and `QSG World` universe. `ABR` is a sample factor code and `QSG World` is a sample universe. To retrieve signal information for your code and universe combination, substitute the factor code in `code` and universe in `universeid`. Also, specify the SEDOL security type and z-score data format. `d` is a table that contains signal information and the `date` and `data` variables.

```
code = 'ABR';
universeid = 'QSG World';
startdate = datetime('today')-10;
enddate = datetime('today');
identifier = 'sedol';
datatype = 'zscore';
d = signals(c,code,universeid,startdate,enddate,identifier,datatype);
```

Access the first few rows of signal information for the first day in the date range by using the `data` variable.

```
data = d.data{1};
head(data)
```

```
ans =

  8×2 table

     sedol       value
```

```
    ________    ________

    'B44WZD'     0.63461
    'B4MG4Z'     0.43807
    '281355'     -3.3183
    'BF4VWH'     0.94079
    'B92SR7'     0.80995
    'BWY4XV'      3.1591
    'B1VZ43'    -0.25296
    '236542'    -0.77368
```

The variables of the resulting table are `sedol` and `value`. The `sedol` variable contains the SEDOL security identifiers. The `value` variable contains the signal information for the corresponding security as a z-score.

Retrieve Monthly Signal Information for Specific Data Format

Using an IHS Markit connection, retrieve monthly signal information using a factor and date range within a specified universe. Specify the SEDOL security type and z-score data format.

Create an IHS Markit connection using your user name and password. `c` is an `ihsmarkitrs` object.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password);
```

Retrieve signal information for the last 3 months using the IHS Markit connection. Specify the `ABR` factor code and `QSG World` universe. `ABR` is a sample factor code and `QSG World` is a sample universe. To retrieve signal information for your code and universe combination, substitute the factor code in `code` and universe in `universeid`. Also, specify the SEDOL security type and z-score data format. `d` is a table that contains signal information and the `date` and `data` variables.

```
code = 'ABR';
universeid = 'QSG World';
startdate = datetime('today')-90;
enddate = datetime('today');
identifier = 'sedol';
datatype = 'zscore';
monthlydata = 'true';
d = signals(c,code,universeid,startdate,enddate, ...
    identifier,datatype,monthlydata);
```

Access the first few rows of signal information for the first month in the date range by using the `data` variable.

```
data = d.data{1};
head(data)
```

```
ans =

  8×2 table

     sedol       value
    ________    ________

    'B44WZD'     0.44178
```

```
    'B4MG4Z'     -1.2075
    '281355'     0.43517
    'BF4VWH'     0.91456
    'B92SR7'       2.065
    '256652'     0.49538
    'B1VZ43'    -0.26471
    'BFRTDG'    -0.69078
```

The variables of the resulting table are `sedol` and `value`. The `sedol` variable contains the SEDOL security identifiers. The `value` variable contains the signal information for the corresponding security as a z-score.

Input Arguments

`c` — IHS Markit connection

`ihsmarkitrs` object

IHS Markit connection, specified as an `ihsmarkitrs` object.

`code` — Factor code

character vector | string scalar

Factor code, specified as a character vector or string scalar.

Example: `"ABR"`

Data Types: `char` | `string`

`universeid` — Universe name

character vector | string scalar

Universe name, specified as a character vector or string scalar.

Example: `'US Total Cap'`

Data Types: `char` | `string`

`startdate` — Start date

`datetime` array | numeric scalar | character vector | string scalar

Start date for a data request, specified as a `datetime` array, numeric scalar, character vector, or string scalar.

Example: `"2017-01-01"`

Data Types: `double` | `char` | `string` | `datetime`

`enddate` — End date

`datetime` array | numeric scalar | character vector | string scalar

End date for a data request, specified as a `datetime` array, numeric scalar, character vector, or string scalar.

Example: `"2017-12-31"`

Data Types: `double` | `char` | `string` | `datetime`

`identifier` — Security type

`"ticker"` (default) | character vector | string scalar | cell array of character vectors | string array

Security type to retrieve, specified as one or more of these values: `'ticker'`, `'cusip'`, or `'sedol'`. You can specify these values as a character vector, string scalar, cell array of character vectors, or string array.

`datatype` — Data format
`"percentile"` (default) | character vector | string scalar

Data format, specified as one of these values.

Data Format Value	Description	Calculation
`"percentile"`	Percentile rank (from `1` through `100`) of the factor	Rank the securities in the universe into percentiles, by using the factor value, in ascending or descending order based on the definition. The `signals` function ranks the securities with the most attractive value as `1` and securities with the least attractive value as `100`.
`"rawratio"`	Raw value of the factor	The numeric output of the factor calculation.
`"rawrank"`	Ordinal rank (from `1` through *n*) of the factor	Rank the securities in the universe in ordinal order, by using the factor value, in ascending or descending order based on the definition.
`"zscore"`	Z-score of the factor	Determine the mean and standard deviation of all factor values in the universe on the specified date. Then, subtract the mean from the factor value of the security and divide the result by the standard deviation.

You can specify each value as a character vector or string scalar.

`monthlydata` — Monthly indicator
`"false"` (default) | character vector | string scalar

Monthly indicator, specified as the value `"true"` or `"false"`. When the `monthlydata` input argument is `"true"`, the `signals` function returns monthly data. Otherwise, the `signals` function returns daily data.

Output Arguments

`d` — Signal information
table

Signal information, returned as a table with the `date` and `data` variables. The `date` variable contains each date in the specified date range. If you specify monthly data using the `monthlydata` input

argument, then the `date` variable contains one row for each month. The `data` variable contains a table of data for each corresponding date. To access the data for the first day in the date range, use dot notation, for example: `d.data{1}`.

Version History

Introduced in R2018b

See Also

`ihsmarkitrs` | `factorgroups` | `factors` | `security` | `universes`

Topics

“Retrieve Factor Rank Data for Portfolio Selection” on page 13-2
“IHS Markit Error Messages” on page 13-4

External Websites

IHS Markit
IHS Markit Research Signals REST Documentation

universes

Retrieve universe information

Syntax

```
d = universes(c)
d = universes(c,universeid)
d = universes(c,universeid,requesttype)
```

Description

`d = universes(c)` returns universe information for all universes using an IHS Markit connection.

`d = universes(c,universeid)` returns universe information for a specific universe.

`d = universes(c,universeid,requesttype)` returns universe information based on the request type.

Examples

Retrieve Universe Information for All Universes

Using an IHS Markit connection, retrieve universe information for all universes.

Create an IHS Markit connection using your user name and password. `c` is an `ihsmarkitrs` object.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password);
```

Retrieve universe information for all universes using the IHS Markit connection. `d` is a table that contains the universe information.

```
d = universes(c);
```

Display information for the first few universes.

```
head(d)
```

```
ans =

  8×6 table

    description                region                universeType    identifier    universeId
    ___________    ___________________________________    ____________    __________    __________

        ''         'Europe, Middle East & Africa'      'Global'          []            248
        ''         '11,12,13,14'                       'Global'          []            227
        ''         '11,12'                             'Global'          []            250
        ''         'Far East'                          'Global'          []            249
        ''         'Far East'                          'Global'          []            228
```

```
        ''          '11,12,13,14'                    'UDM'             []             324
        ''          '11,12,13,14'                    'Global'          []            1552
        ''          '11,12,13,14'                    'Global'          []            1293
```

The variables are:

- `description` — Universe description
- `region` — Country or region code
- `universeType` — Universe type
- `identifier` — Identification type
- `universeId` — Universe identifier
- `universe` — Universe name

Retrieve Universe Information for Specific Universe

Using an IHS Markit connection, retrieve universe information for a specific universe.

Create an IHS Markit connection using your user name and password. `c` is an `ihsmarkitrs` object.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password);
```

Retrieve universe information for the `QSG World` universe using the IHS Markit connection. `d` is a table that contains the universe information.

```
universeid = "QSG World";
d = universes(c,universeid)
```

```
d =

  1×6 table

    description       region        universeType    identifier    universeId     universe
    ___________    _____________    ____________    __________    __________    ___________

        ''         '11,12,13,14'      'Global'       'Sedol'         133        'QSG World'
```

The variables are:

- `description` — Universe description
- `region` — Country or region code
- `universeType` — Universe type
- `identifier` — Identification type
- `universeId` — Universe identifier
- `universe` — Universe name

Retrieve Universe Information for Specific Request Type

Using an IHS Markit connection, retrieve universe information for a specific universe. Specify a historical request to retrieve historical information for the universe.

Create an IHS Markit connection using your user name and password. `c` is an `ihsmarkitrs` object.

```
username = 'ABCDEF';
password = 'ABC123';
c = ihsmarkitrs(username,password);
```

Retrieve universe information for the `QSG World` universe using the IHS Markit connection. Specify retrieving historical information by using the `HistoryDetail` request type. `d` is a table that contains the historical universe information.

```
universeid = "QSG World";
requesttype = 'HistoryDetail';
d = universes(c,universeid,requesttype)
```

```
d =

  2×4 table

     universe      freqType      startDate        endDate
    ___________    _________    ____________    ____________

    'QSG World'    'Daily'      '03/22/2007'    '03/14/2018'
    'QSG World'    'Monthly'    '12/30/1988'    '05/31/2017'
```

The variables are:

- `universe` — Universe name
- `freqType` — Data frequency (or periodicity)
- `startDate` — Start date of the life of the universe
- `endDate` — End date of the life of the universe

Input Arguments

c — IHS Markit connection

`ihsmarkitrs` object

IHS Markit connection, specified as an `ihsmarkitrs` object.

universeid — Universe name

character vector | string scalar

Universe name, specified as a character vector or string scalar.

Example: `'US Total Cap'`

Data Types: `char` | `string`

requesttype — Request type

character vector | string scalar

Request type, specified as the value `'HistoryDetail'`, `'Mapping'`, or `'Country'`. Use the `'HistoryDetail'` value to return historical information from the universe that you specify using the

`universeid` input argument. Use the `'Mapping'` value to return a list of the factors in the specified universe. Use the `'Country'` value to return the country identifiers that apply to the specified universe.

You can specify each value as a character vector or string scalar.

Output Arguments

d — Universe information

table

Universe information, returned as a table. The following table describes the variables in the returned data. (The variables vary depending on the request type that you specify in the `requesttype` input argument.)

Variable Name	Description	Data Type
`universeId`	Universe identifier	`double`
`description`	Universe description	cell array of character vectors
`region`	Country or region code	cell array of character vectors
`universeType`	Universe type	cell array of character vectors
`identifier`	Identification type	cell array of character vectors
`universe`	Universe name	cell array of character vectors
`freqType`	Data frequency (or periodicity)	cell array of character vectors
`startDate`	Start date of the life of the universe	cell array of character vectors
`endDate`	End date of the life of the universe	cell array of character vectors
`data`	Factors in the universe	structure
`country`	Countries that apply to the universe	cell array of character vectors

Version History

Introduced in R2018b

See Also

`ihsmarkitrs` | `factorgroups` | `factors` | `security` | `signals`

Topics

“Retrieve Factor Rank Data for Portfolio Selection” on page 13-2
“IHS Markit Error Messages” on page 13-4

External Websites

IHS Markit
IHS Markit Research Signals REST Documentation

fds

FactSet Workstation connection

Description

The `fds` function creates an `fds` object. The `fds` object represents a FactSet Workstation connection.

After you create an `fds` object, you can use the object functions to retrieve real-time data for securities. For credentials, contact FactSet Research Systems.

Creation

Syntax

```
c = fds(username,password)
c = fds(username,password,finfo)
```

Description

`c = fds(username,password)` creates a FactSet Workstation connection using a user name and password. By default, this syntax uses the field information file `rt_fields.xml`, which is found on the MATLAB path.

`c = fds(username,password,finfo)` creates a connection using the specified field information file.

Input Arguments

`username` — FactSet user name
character vector | string scalar

FactSet user name, specified as a character vector or string scalar. To find your user name, contact FactSet Research Systems.

Example: `'ABCD_EFGH_IJKL'`

Data Types: `char` | `string`

`password` — FactSet password
character vector | string scalar

FactSet password, specified as a character vector or string scalar. To find your password, contact FactSet Research Systems.

Example: `'XXXXXXXX'`

Data Types: `char` | `string`

`finfo` — Field information file
character vector | string scalar

Field information file, specified as a character vector or string scalar. To obtain the field information file, contact FactSet Research Systems. Specify the full file path to the field information file.

Example: `'C:\Program Files (x86)\FactSet\FactSetDataFeed\fdsrt-2\etc\rt_fields.xml'`

Data Types: `char` | `string`

Properties

`Handle` — FactSet handle

handle object

FactSet handle, specified as a handle object.

Example: `[1×1 COM.FDSRTCom_FDF]`

Object Functions

realtime	Obtain real-time data from FactSet Workstation
stop	Cancel real-time request
close	Disconnect from FactSet Workstation

Examples

Create FactSet Workstation Connection

Create a FactSet Workstation connection. Then, retrieve real-time data for a security.

Connect to the FactSet Workstation using a user name and password. By default, the `fds` function uses the field information file `rt_fields.xml`, which is found on the MATLAB path. `c` is an `fds` object.

```
username = 'ABCD_EFGH_IJKL';
password = 'XXXXXXXX';

c = fds(username,password)
```

```
c =

  fds with properties:

    Handle: [1×1 COM.FDSRTCom_FDF]
```

Retrieve real-time data for the `FDS1` service and `ABCD-USA` security by using the FactSet Workstation connection. Use the default event handler function `myMessageEventHandler` to process real-time data events from the FactSet Workstation. To access the code for the default event handler function, enter `edit myMessageEventHandler` at the command line. You can write a custom function to process real-time data events differently. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
Srv = 'FDS1';
Sec = 'ABCD-USA';
Cb = @(varargin)myMessageEventHandler(varargin);
t = realtime(c,Srv,Sec,Cb)
```

```
t =

     1

ABCD-USA:D 11-Sep-2017 14:04:53 6.27
ABCD-USA:D 11-Sep-2017 14:07:00 6.29
...
```

The `realtime` function returns a data tag `t` for the real-time request. Then, the event handler function returns the following data to the Command Window:

- Security name
- Date
- Time
- Last price

Stop real-time data retrieval.

```
stop(c,t)
```

Close the FactSet Workstation connection.

```
close(c)
```

Create FactSet Workstation Connection Using Field Information File

Create a FactSet Workstation connection and specify the field information file. Then, retrieve real-time data for a security.

Connect to the FactSet Workstation using a user name, password, and field information file. `c` is an `fds` object.

```
username = 'ABCD_EFGH_IJKL';
password = 'XXXXXXXX';
finfo = 'C:\Program Files (x86)\FactSet\FactSetDataFeed\fdsrt-2\etc\rt_fields.xml';

c = fds(username,password,finfo)

c =

  fds with properties:

    Handle: [1×1 COM.FDSRTCom_FDF]
```

Retrieve real-time data for the `FDS1` service and `ABCD-USA` security by using the FactSet Workstation connection. Use the default event handler function `myMessageEventHandler` to process real-time data events from the FactSet Workstation. To access the code for the default event handler function, enter `edit myMessageEventHandler` at the command line. You can write a custom function to process real-time data events differently. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
Srv = 'FDS1';
Sec = 'ABCD-USA';
Cb = @(varargin)myMessageEventHandler(varargin);
t = realtime(c,Srv,Sec,Cb)
```

```
t =

     1

ABCD-USA:D 11-Sep-2017 14:04:53 6.27
ABCD-USA:D 11-Sep-2017 14:07:00 6.29
...
```

The `realtime` function returns a data tag `t` for the real-time request. Then, the event handler function returns the following data to the Command Window:

- Security name
- Date
- Time
- Last price

Stop real-time data retrieval.

```
stop(c,t)
```

Close the FactSet Workstation connection.

```
close(c)
```

Version History

Introduced in R2013a

See Also

Topics

“Writing and Running Custom Event Handler Functions” on page 1-26

External Websites

FactSet Research Systems

realtime

Obtain real-time data from FactSet Workstation

Syntax

```
T = realtime(c,Srv,Sec,Cb)
T = realtime(c,Srv,Sec)
```

Description

`T = realtime(c,Srv,Sec,Cb)` asynchronously requests real-time or streaming data from the FactSet Workstation.

`T = realtime(c,Srv,Sec)` asynchronously requests real-time or streaming data from the FactSet Workstation. When `Cb` is not specified, the default message event handler `factsetMessageEventHandler` is used.

Examples

Request FactSet Workstation Real-Time Data with User-Defined Event Handler

To request real-time or streaming data for the symbol `'ABDC-USA'` from the service `'FDS1'`, a user-defined event handler (`myMessageEventHandler`) is used to process message events using this syntax.

```
t = realtime(c,'FDS1','ABCD-USA',@(varargin)myMessageEventHandler(varargin))
```

Request FactSet Workstation Real-Time Data Using Default Event Handler

To request real-time or streaming data for the symbol `'ABDC-USA'` from the service `'FDS1'`, using this syntax.

```
t = realtime(c,'FDS1','ABCD-USA')
```

The default event handler is used which returns a structure `X` to the base MATLAB workspace containing the latest data for the symbol `'ABCD-USA'`. `X` is updated as new message events are received.

Input Arguments

`c` — FactSet Workstation connection

connection object

FactSet Workstation connection, specified as a connection object created using `fds`.

`Srv` — Data source or supplier

character vector | string scalar

Data source or supplier, specified as a character vector or string scalar.

Example: `'FDS1'`

Data Types: `char` | `string`

`Sec` — Security symbol
character vector | string scalar

Security symbol, specified as a character vector or string scalar.

Example: `'ABCD-USA'`

Data Types: `char` | `string`

`Cb` — Event handler
function handle

Event handler, specified as a function handle requests real-time or streaming data from the service FactSet Workstation.

If `Cb` is not specified, the default message event handler `factsetMessageEventHandler` is used.

Example: `@(varargin)myMessageEventHandler(varargin)`

Data Types: `function_handle`

Output Arguments

`T` — Real-time data tag
nonnegative integer

Real-time data tag, returned as a nonnegative integer from FactSet Workstation.

Version History

Introduced in R2013a

See Also

`fds` | `close` | `stop`

Topics

“Writing and Running Custom Event Handler Functions” on page 1-26

stop

Cancel real-time request

Syntax

```
stop(c,T)
```

Description

`stop(c,T)` cancels a real-time request. This function cleans up resources associated with real-time requests that are no longer needed.

Examples

Cancel FactSet Workstation Real-Time Request

Terminate a FactSet Workstation real-time request.

```
T = realtime(c,'FDS1','GOOG-USA')
stop(c,T)
```

Input Arguments

c — FactSet Workstation connection

connection object

FactSet Workstation connection, specified as a connection object created using `fds`.

T — Real-time request tag

nonnegative integer

Real-time request tag, specified using `realtime`.

Data Types: `double`

Version History

Introduced in R2013a

See Also

`fds` | `close` | `realtime`

close

Disconnect from FactSet Workstation

Syntax

```
close(c)
```

Description

`close(c)` disconnects from the FactSet Workstation given the connection object, `F`.

Examples

Close FactSet Workstation Connection

Close the FactSet Workstation connection.

```
T = realtime(c,'FDS1','GOOG-USA')
close(c)
```

Input Arguments

`c` — FactSet Workstation connection

connection object

FactSet Workstation connection, specified as a connection object created using `fds`.

Version History

Introduced in R2013a

See Also

`fds` | `realtime` | `stop`

fred

Connect to FRED data servers

Description

The `fred` function creates a `fred` object. The `fred` object represents a FRED connection.

After you create a `fred` object, you can use the object functions to retrieve economic data for a FRED series. You can also retrieve data for a specific date or date range.

Creation

Syntax

```
c = fred
c = fred(url)
```

Description

`c = fred` returns a FRED connection to the FRED data server using the default URL `'https://fred.stlouisfed.org/'`.

`c = fred(url)` returns a FRED connection using the specified URL.

Input Arguments

`url` — URL of FRED data server
character vector | string scalar

URL of the FRED data server, specified as a character vector or string scalar.

Example: `'https://fred.stlouisfed.org/'`

Data Types: `char` | `string`

Properties

`URL` — URL of FRED data server
character vector

URL of the FRED data server, specified as a character vector.

The `fred` function sets this property using the `url` input argument.

Example: `'https://fred.stlouisfed.org/'`

Data Types: `char`

`IP` — IP address
`[]` (default) | character vector

IP address of the proxy server, specified as a character vector.

Data Types: `char`

`Port` — Port number

`[]` (default) | numeric scalar

Port number of the proxy server, specified as a numeric scalar.

Data Types: `double`

`DataReturnFormat` — Data return format

`[]` (default) | `'table'` | `'timetable'`

Data return format, specified as one of these values, which determine the data type of the returned data.

Value	Data Type of Returned Data
`[]` (default)	`structure`
`'table'`	`table`
`'timetable'`	`timetable`

You can specify these values using a character vector or string (for example, `"table"`).

When you create a `fred` object, the `fred` function leaves this property unset. Set this property value manually at the command line or in a script using dot notation, for example:

```
c.DataReturnFormat = 'table';
```

After setting the `DataReturnFormat` property, use the `fetch` function to retrieve data.

`DatetimeType` — Date and time data type

`[]` (default) | `'datetime'`

Date and time data type, specified as one of these values.

Value	Data Type of Returned Data
`[]` (default)	MATLAB date numbers
`'datetime'`	`datetime` array

You can specify these values using a character vector or string (for example, `"datetime"`).

When you create a `fred` object, the `fred` function leaves this property unset. Set this property value manually at the command line or in a script using dot notation, for example:

```
c.DatetimeType = 'datetime';
```

After setting the `DatetimeType` property, use the `fetch` function to retrieve data.

Object Functions

fetch	Request data from FRED data servers
isconnection	Determine if connections to FRED data servers are valid
close	Close connections to FRED data servers

Examples

Connect to FRED

Connect to the FRED® data server, and then retrieve historical data for a series.

Connect to the FRED data server.

```
c = fred
```

`c` is a FRED connection with these properties:

- URL for the FRED data server
- IP address of the proxy server
- Port number of the proxy server
- Date and time data type for returned data
- Data return format for returned data

Retrieve the `ip` property of the FRED connection `c`.

```
c.IP
```

Retrieve the `port` property of the FRED connection `c`.

```
c.Port
```

Adjust the display data format for currency.

```
format bank
```

Retrieve all historical data for the US / Euro Foreign Exchange Rate series. `d` contains the series description.

```
series = 'DEXUSEU';
```

```
d = fetch(c,series);
```

Close the FRED connection.

```
close(c)
```

Connect to FRED with URL

Connect to the FRED® data server using a URL, and then retrieve historical data for a series.

Connect to the FRED data server using the URL `'https://fred.stlouisfed.org/'`.

```
url = 'https://fred.stlouisfed.org/';
c = fred(url)
```

`c` is a FRED connection with these properties:

- URL for the FRED data server
- IP address of the proxy server
- Port number of the proxy server
- Date and time data type for returned data
- Data return format for returned data

Retrieve the `ip` property of the FRED connection `c`.

```
c.IP
```

Retrieve the `port` property of the FRED connection `c`.

```
c.Port
```

Adjust the display data format for currency.

```
format bank
```

Retrieve all historical data for the US / Euro Foreign Exchange Rate series. `d` contains the series description.

```
series = 'DEXUSEU';
```

```
d = fetch(c,series);
```

Close the FRED connection.

```
close(c)
```

Version History

Introduced in R2006b

See Also

Topics

“Retrieve Historical Data Using FRED” on page 1-10

close

Close connections to FRED data servers

Syntax

```
close(c)
```

Arguments

c	FRED connection object created with `fred`.

Description

`close(c)` closes the connection to the FRED data server.

Examples

Close FRED® Connection

Connect to the data server at the URL `https://research.stlouisfed.org/fred2/`.

```
c = fred('https://fred.stlouisfed.org/');
```

Adjust the display data format for currency.

```
format bank
```

Retrieve all historical data for the US / Euro Foreign Exchange Rate series.

```
series = 'DEXUSEU';
```

```
d = fetch(c,series);
```

`d` contains the series description.

Close the FRED® connection.

```
close(c)
```

Version History

Introduced in R2006b

See Also

`fred`

Topics

“Retrieve Historical Data Using FRED” on page 1-10

fetch

Request data from FRED data servers

Syntax

```
d = fetch(c,series)
d = fetch(c,series,date)
d = fetch(c,series,startdate,enddate)
```

Description

`d = fetch(c,series)` returns FRED data using the FRED connection `c` and the specified FRED series.

`d = fetch(c,series,date)` returns FRED data for a specific date.

`d = fetch(c,series,startdate,enddate)` returns FRED data for the date range from `startdate` through `enddate`.

Examples

Fetch All Available FRED® Data

Connect to the FRED® data server using the URL `'https://fred.stlouisfed.org/'`.

```
url = 'https://fred.stlouisfed.org/';
c = fred(url);
```

Fetch all available daily foreign exchange rates between the US dollar and the Euro using the series `'DEXUSEU'`.

```
series = 'DEXUSEU';
d = fetch(c,series)
```

```
d =

  struct with fields:

                Title: ' U.S. / Euro Foreign Exchange Rate'
             SeriesID: ' DEXUSEU'
               Source: ' Board of Governors of the Federal Reserve System (US)'
              Release: ' H.10 Foreign Exchange Rates'
    SeasonalAdjustment: ' Not Seasonally Adjusted'
            Frequency: ' Daily'
                Units: ' U.S. Dollars to One Euro'
            DateRange: ' 1999-01-04 to 2017-02-03'
          LastUpdated: ' 2017-02-06 3:52 PM CST'
                Notes: ' Noon buying rates in New York City for cable transfers payable in forei
                 Data: [4720×2 double]
```

`d.Data` is an N-by-2 double array that contains dates in the first column and the series values in the second column.

Close the FRED® connection.

```
close(c)
```

Fetch FRED® Data for Date

Connect to the FRED® data server using the URL `'https://fred.stlouisfed.org/'`.

```
url = 'https://fred.stlouisfed.org/';
c = fred(url);
```

Adjust the display data format for currency.

```
format bank
```

Fetch data for a day three months ago using the series `'DTB6'`.

```
series = 'DTB6';
date = floor(now)-90;
d = fetch(c,series,date)
```

```
d =

  struct with fields:

                Title: ' 6-Month Treasury Bill: Secondary Market Rate'
             SeriesID: ' DTB6'
               Source: ' Board of Governors of the Federal Reserve System (US)'
              Release: ' H.15 Selected Interest Rates'
    SeasonalAdjustment: ' Not Seasonally Adjusted'
            Frequency: ' Daily'
                Units: ' Percent'
            DateRange: ' 1958-12-09 to 2017-02-06'
          LastUpdated: ' 2017-02-07 3:51 PM CST'
                Notes: ' Discount Basis'
                 Data: [736644.00 0.58]
```

`d.Data` is an N-by-2 double array that contains the date in the first column and the series value in the second column.

Close the FRED® connection.

```
close(c)
```

Fetch FRED® Data for Date Range

Connect to the FRED® data server using the URL `'https://fred.stlouisfed.org/'`.

```
url = 'https://fred.stlouisfed.org/';
c = fred(url);
```

Adjust the display data format for currency.

```
format bank
```

Retrieve historical data for the US / Euro Foreign Exchange Rate series.

```
series = 'DEXUSEU';
```

Fetch five months of data from January 1, 2007 through June 1, 2007.

```
startdate = '01/01/2007';
enddate = '06/01/2007';
d = fetch(c,series,startdate,enddate)
```

```
d =

  struct with fields:

                Title: ' U.S. / Euro Foreign Exchange Rate'
             SeriesID: ' DEXUSEU'
               Source: ' Board of Governors of the Federal Reserve System (US)'
              Release: ' H.10 Foreign Exchange Rates'
    SeasonalAdjustment: ' Not Seasonally Adjusted'
            Frequency: ' Daily'
                Units: ' U.S. Dollars to One Euro'
            DateRange: ' 1999-01-04 to 2017-02-03'
          LastUpdated: ' 2017-02-06 3:52 PM CST'
                Notes: ' Noon buying rates in New York City for cable transfers payable in forei
                 Data: [110×2 double]
```

`d.Data` is an N-by-2 double array that contains dates in the first column and the series values in the second column.

Close the FRED® connection.

```
close(c)
```

Retrieve Available FRED Data as Table

Connect to the FRED data server using the URL `'https://fred.stlouisfed.org/'`.

```
url = 'https://fred.stlouisfed.org/';
c = fred(url);
```

Adjust the display format for currency.

```
format bank
```

Set the data return format to table using the `DataReturnFormat` property of the `fred` object.

```
c.DataReturnFormat = 'table';
```

Fetch all available daily foreign exchange rates between the US dollar and the Euro using the series `'DEXUSEU'`. The table `d` contains one row for the series. Each variable describes a piece of information about the series.

```
series = 'DEXUSEU';
d = fetch(c,series)

d=1×11 table
                  Title                       SeriesID                                Source
    ____________________________________    _________    ____________________________________

    ' U.S. / Euro Foreign Exchange Rate'    ' DEXUSEU'    ' Board of Governors of the Federal Res
```

Access the data in the series from the `Data` variable.

```
data = d.Data{1};
```

Display the first few foreign exchange rates. The first variable in the table is the date and the second variable is the foreign exchange rate. Also, the first variable is an array of MATLAB® date numbers.

```
head(data)

ans=8×2 table
      Var1       Var2
    _________    ____

    730124.00    1.18
    730125.00    1.18
    730126.00    1.16
    730127.00    1.17
    730128.00    1.16
    730131.00    1.15
    730132.00    1.15
    730133.00    1.17
```

Close the FRED connection.

```
close(c)
```

Retrieve Available FRED Data as Table with Datetime

Connect to the FRED data server using the URL `'https://fred.stlouisfed.org/'`.

```
url = 'https://fred.stlouisfed.org/';
c = fred(url);
```

Adjust the display format for currency.

```
format bank
```

Set the data return format to table using the `DataReturnFormat` property of the `fred` object. Also, set the date and time format to `datetime` array using the `DatetimeType` property.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Fetch all available daily foreign exchange rates between the US dollar and the Euro using the series `'DEXUSEU'`. The table `d` contains one row for the series. Each variable describes a piece of information about the series.

```
series = 'DEXUSEU';
d = fetch(c,series)
```

```
d=1×11 table
                    Title                       SeriesID                                Source
    ____________________________________    ____________    ____________________________________

    ' U.S. / Euro Foreign Exchange Rate'    ' DEXUSEU'    ' Board of Governors of the Federal Res
```

Access the data in the series from the `Data` variable.

```
data = d.Data{1};
```

Display the first few foreign exchange rates. The first variable in the table is the date and the second variable is the foreign exchange rate. Also, the first variable is a `datetime` array.

```
head(data)
```

```
ans=8×2 table
            Var1            Var2
    ____________________    ____

    04-Jan-1999 00:00:00    1.18
    05-Jan-1999 00:00:00    1.18
    06-Jan-1999 00:00:00    1.16
    07-Jan-1999 00:00:00    1.17
    08-Jan-1999 00:00:00    1.16
    11-Jan-1999 00:00:00    1.15
    12-Jan-1999 00:00:00    1.15
    13-Jan-1999 00:00:00    1.17
```

Close the FRED connection.

```
close(c)
```

Input Arguments

`c` — FRED connection

connection object

FRED connection, specified as a connection object created using `fred`.

`series` — FRED series

character vector | string scalar

FRED series, specified as a character vector or string scalar.

Example: `'DEXUSEU'`

Data Types: `char` | `string`

`date` — Date
`datetime` | matrix | character vector | string scalar

Date, specified as a `datetime` value, matrix, character vector, or string scalar. For details about the data types, see `datetime`.

Data Types: `double` | `char` | `cell` | `datetime` | `string`

`startdate` — Start date
`datetime` | matrix | character vector | string scalar

Start date in a date range, specified as a `datetime` value, matrix, character vector, or string scalar. For details about the data types, see `datetime`.

Data Types: `double` | `char` | `cell` | `datetime` | `string`

`enddate` — End date
`datetime` | matrix | character vector | string scalar

End date in a date range, specified as a `datetime` value, matrix, character vector, or string scalar. For details about the data types, see `datetime`.

Data Types: `double` | `char` | `cell` | `datetime` | `string`

Output Arguments

`d` — FRED data
structure

FRED data, returned as a structure. For details about FRED data, see https://fred.stlouisfed.org/.

Version History

Introduced in R2006b

See Also

`close` | `isconnection` | `fred`

Topics
“Retrieve Historical Data Using FRED” on page 1-10

get

Retrieve properties of FRED connection objects

Syntax

```
value = get(c,'PropertyName')
value = get(c)
```

Arguments

`c`	FRED connection object created with `fred`.
`'PropertyName'`	A MATLAB character vector, string, cell array of character vectors, or string array containing property names. Property names are: • `'url'` • `'ip'` • `'port'`

Description

`value = get(c,'PropertyName')` returns a MATLAB structure containing the value of the specified properties for the FRED connection object.

`value = get(c)` returns the value for all properties.

Examples

Establish a connection, `c`, to a FRED data server.

```
c = fred('https://fred.stlouisfed.org/')
```

Retrieve the port and IP address for the connection.

```
p = get(c,{'port','ip'})
p =
     port: 8194
     ip: 111.222.33.444
```

Version History

Introduced in R2006b

See Also

`close` | `fetch` | `isconnection`

Topics

“Retrieve Historical Data Using FRED” on page 1-10

isconnection

Determine if connections to FRED data servers are valid

Syntax

```
x = isconnection(c)
```

Arguments

`c`	FRED connection object created with `fred`.

Description

`x = isconnection(c)` returns `x = 1` if a connection to the FRED data server is valid, and `x = 0` otherwise.

Examples

Verify FRED® Connection

Establish a connection `c` to a FRED® data server.

```
c = fred('https://fred.stlouisfed.org/');
```

Verify that `c` is a valid connection.

```
x = isconnection(c)
```

```
x =

     1
```

Adjust the display data format for currency.

```
format bank
```

Retrieve all historical data for the US / Euro Foreign Exchange Rate series.

```
series = 'DEXUSEU';

d = fetch(c,series);
```

`d` contains the series description.

Close the FRED® connection.

```
close(c)
```

Version History

Introduced in R2006b

See Also

`close` | `fetch` | `fred`

Topics

“Retrieve Historical Data Using FRED” on page 1-10

haver

Connect to local Haver Analytics database

Description

The `haver` function creates a `haver` object. The `haver` object represents a Haver Analytics database connection.

After you create a `haver` object, you can use the object functions to retrieve all historical data for a variable. You can also retrieve historical data in a date range.

Creation

Syntax

```
c = haver(databasename)
```

Description

`c = haver(databasename)` establishes a connection to a Haver Analytics database and sets the "DatabaseName" on page 15-0 property.

Requirement: You need both read and write permissions on the database file to establish a connection. Otherwise, this error message appears at the command line: `Unable to open specified database file`.

Input Arguments

`databasename` — Database name
character vector | string scalar

Database name, specified as a character vector or string scalar. The database name is the full path to the Haver Analytics database file on the local machine.

Example: `'C:\haver\haverd.dat'`

Data Types: `char` | `string`

Properties

`DatabaseName` — Database name
character vector

This property is read-only.

Database name, specified as a character vector. The database name is the full path to the Haver Analytics database file on the local machine.

The `haver` function sets this property using the `databasename` input argument.

Example: `'C:\haver\haverd.dat'`

Data Types: `char`

`DataReturnFormat` — Data return format
`''` (default) | `'table'` | `'timetable'`

Data return format, specified as one of these values, which determine the data type of the returned data.

Value	Data Type of Returned Data
`''` (default)	`structure` or numeric array
`'table'`	`table`
`'timetable'`	`timetable`

You can specify these values using a character vector or string (for example, `"table"`).

When you create a `haver` object, the `haver` function leaves this property unset. Set this property value manually at the command line or in a script using dot notation, for example:

```
c.DataReturnFormat = 'table';
```

The default data type value depends on the function. The following table describes the default value for each corresponding supported function.

Supported Function	Default Data Type
`fetch`	numeric array
`info`	`structure`
`nextinfo`	`structure`

`DatetimeType` — Date and time data type
`''` (default) | `'datetime'`

Date and time data type, specified as one of these values.

Value	Data Type of Returned Data
`''` (default)	MATLAB date numbers or character vectors
`'datetime'`	`datetime` array

You can specify these values using a character vector or string (for example, `"datetime"`).

When you create a `haver` object, the `haver` function leaves this property unset. Set this property value manually at the command line or in a script using dot notation, for example:

```
c.DatetimeType = 'datetime';
```

The default data type value depends on the function. The following table describes the default value for each corresponding supported function.

Supported Function	Default Data Type
`fetch`	numeric scalar
`info`	character vector
`nextinfo`	character vector

Note If you leave the DataReturnFormat property set to the default value and you retrieve data using the `fetch` function, the date and time data type remains a numeric scalar. To retrieve date and time values as a `datetime` array, set the `DataReturnFormat` property to `'timetable'`, or set the `DataReturnFormat` property to `'table'` and the `DatetimeType` property to `'datetime'`.

Object Functions

close	Close Haver Analytics database
fetch	Request data from Haver Analytics database
get	Retrieve properties from Haver Analytics connection objects
info	Retrieve information about Haver Analytics variables
isconnection	Determine if connections to Haver Analytics data servers are valid
nextinfo	Retrieve information about next Haver Analytics variable

Examples

Connect to Haver Analytics Database

Create a Haver Analytics database connection. Then, retrieve historical data for a specified date range.

Create the Haver Analytics database connection using the local Haver Analytics database file. `c` is a `haver` object.

```
databasename = 'C:\haver\usecon.dat';
c = haver(databasename)
```

```
c =

  haver with properties:

        DatabaseName: 'C:\haver\usecon.dat'
        DatetimeType: ''
    DataReturnFormat: ''
```

Adjust the display format for currency.

```
format bank
```

Retrieve historical data from January 1, 2005, through December 31, 2005, for `'FFED'`.

```
variable = 'FFED'; % return data for FFED
startdate = '01/01/2005'; % start of date range
enddate = '12/31/2005';   % end of date range

d = fetch(c,variable,startdate,enddate);
```

Display the first three rows of data. `d` contains the numeric representation of the date in the first column and the closing value in the second column.

```
d(1:3,:)

ans =

     732315.00          2.25
     732316.00          2.25
     732317.00          2.25
```

Close the Haver Analytics connection.

```
close(c)
```

Version History

Introduced in R2007a

See Also

Topics

“Retrieve Historical Data Using Haver Analytics” on page 1-12

aggregation

Set Haver Analytics aggregation mode

Syntax

```
X = aggregation (C)
X = aggregation (C,V)
```

Description

`X = aggregation (C)` returns the current aggregation mode.

`X = aggregation (C,V)` sets the current aggregation mode to `V`. The following table lists possible values for `V`.

Value of `V`	Aggregation mode	Behavior of `aggregation` function
`0`	`strict`	`aggregation` does not fill in values for missing data.
`1`	`relaxed`	`aggregation` fills in missing data based on data available in the requested period.
`2`	`forced`	`aggregation` fills in missing data based on some past value.
`-1`	Not recognized	`aggregation` resets `V` to its last valid setting.

Version History

Introduced in R2008b

See Also

`haver` | `close` | `fetch` | `info` | `isconnection` | `nextinfo`

Topics

“Retrieve Historical Data Using Haver Analytics” on page 1-12

close

Close Haver Analytics database

Syntax

```
close(c)
```

Arguments

`c`	Haver Analytics connection object created with `haver`.

Description

`close(c)` closes the connection to the Haver Analytics database.

Examples

Establish a connection to a Haver Analytics database:

```
c = haver('d:\work\haver\data\haverd.dat')
```

Close the connection:

```
close(c)
```

Version History

Introduced in R2007a

See Also

`haver`

Topics

“Retrieve Historical Data Using Haver Analytics” on page 1-12

fetch

Request data from Haver Analytics database

Syntax

```
d = fetch(c,variable)
d = fetch(c,variable,startdate,enddate)
d = fetch(c,variable,startdate,enddate,period)
```

Description

`d = fetch(c,variable)` returns historical data for the Haver Analytics variable `s`, using the connection object `c`.

`d = fetch(c,variable,startdate,enddate)` returns historical data between the dates `startdate` and `enddate`.

`d = fetch(c,variable,startdate,enddate,period)` returns historical data in time periods specified by `period`.

Examples

Retrieve Variable Data

Connect to the Haver Analytics database.

```
c = haver('c:\work\haver\usecon.dat');
```

Retrieve all historical data for the Haver Analytics variable `'FFED'`. The descriptor for this variable is Federal Funds [Effective] Rate (% p.a.).

```
variable = 'FFED'; % return data for FFED

d = fetch(c,variable);
```

Display the first three rows of data.

```
d(1:3,:)

ans =

     715511.00          2.38
     715512.00          2.50
     715515.00          2.50
```

`d` contains the numeric representation of the date in the first column and the closing value in the second column.

Close the Haver Analytics connection.

```
close(c)
```

Retrieve Variable Data for Specified Date Range

Connect to the Haver Analytics database.

```
c = haver('c:\work\haver\usecon.dat');
```

Retrieve historical data from January 1, 2005, through December 31, 2005, for `'FFED'`.

```
variable = 'FFED'; % return data for FFED
startdate = '01/01/2005'; % start of date range
enddate = '12/31/2005';   % end of date range

d = fetch(c,variable,startdate,enddate);
```

Display the first three rows of data.

```
d(1:3,:)

ans =

     732315.00          2.25
     732316.00          2.25
     732317.00          2.25
```

`d` contains the numeric representation of the date in the first column and the closing value in the second column.

Close the Haver Analytics connection.

```
close(c)
```

Retrieve Quarterly Data for Specified Date Range

Connect to the Haver Analytics database.

```
c = haver('c:\work\haver\usecon.dat');
```

Retrieve the information of the Haver Analytics variable `'FFED'`. The descriptor for this variable is Federal Funds [Effective] Rate (% p.a.).

```
variable = 'FFED';

x = info(c,variable);
```

`info` returns the structure `x` containing fields describing the Haver Analytics variable.

Retrieve quarterly data. When you specify a date that is outside the date range in the variable, you might experience unexpected results. To prevent this, use the `EndDate` field for the end of the date range.

```
startdate = '06/01/2000';  % start of date range
enddate = x.EndDate;       % end of date range
period = 'q';              % quarterly data
```

```
d = fetch(c,variable,startdate,enddate,period)
```

Display the first three rows of data.

```
d(1:3,:)
```

```
ans =

     730759.00          6.52
     730851.00          6.50
     730941.00          5.61
```

`d` contains the numeric representation of the date in the first column and the closing value in the second column.

Close the Haver Analytics connection.

```
close(c)
```

Retrieve Variable Data as Table

Connect to the Haver Analytics database.

```
c = haver('c:\work\haver\usecon.dat');
```

Adjust the display format for currency.

```
format bank
```

Set the data return format to table using the `DataReturnFormat` property of the `haver` object.

```
c.DataReturnFormat = 'table';
```

Retrieve historical data from January 1, 2005, through December 31, 2005, for `'ABQ'` and display the results. `ABQ` provides bankruptcy filings in the US.

```
variable = 'ABQ'; % return data for ABQ
startdate = '01/01/2005'; % start of date range
enddate = '12/31/2005';   % end of date range

d = fetch(c,variable,startdate,enddate)
```

```
ans =

  4×2 table

      Time       TotalBankruptcyFilings_U_S__Units_
    _________    __________________________________

    732402.00                401149.00
    732493.00                467333.00
    732585.00                542002.00
    732677.00                667431.00
```

`d` is a table with the date in the first variable and the bankruptcy amount in the second variable. The date is an array of MATLAB date numbers.

Close the Haver Analytics connection.

```
close(c)
```

Retrieve Variable Data as Table with Datetime

Connect to the Haver Analytics database.

```
c = haver('c:\work\haver\usecon.dat');
```

Adjust the display format for currency.

```
format bank
```

Set the data return format to table using the `DataReturnFormat` property of the `haver` object. Also, set the date and time return format to a `datetime` array using the `DatetimeType` property.

```
c.DataReturnFormat = 'table';
c.DatetimeType = 'datetime';
```

Retrieve historical data from January 1, 2005, through December 31, 2005, for `'ABQ'` and display the results. `ABQ` provides bankruptcy filings in the US.

```
variable = 'ABQ'; % return data for ABQ
startdate = '01/01/2005'; % start of date range
enddate = '12/31/2005';   % end of date range

d = fetch(c,variable,startdate,enddate)
```

```
ans =

  4×2 table

           Time            TotalBankruptcyFilings_U_S__Units_
    ____________________    __________________________________

    31-Mar-2005 00:00:00                401149.00
    30-Jun-2005 00:00:00                467333.00
    30-Sep-2005 00:00:00                542002.00
    31-Dec-2005 00:00:00                667431.00
```

`d` is a table with the date and time in the first variable and the bankruptcy amount in the second variable. The first variable is a `datetime` array.

Close the Haver Analytics connection.

```
close(c)
```

Input Arguments

c — Haver Analytics connection

connection object

Haver Analytics connection, specified as a connection object created using `haver`.

`variable` — Haver Analytics variable
character vector | string scalar | cell array of character vectors | string array

Haver Analytics variable, specified as a character vector, string scalar, cell array of character vectors, or string array to denote which historical data to retrieve.

Example: `'FFED'`

Data Types: `char` | `string` | `cell`

`startdate` — Start date
character vector | string scalar | MATLAB date number

Start date, specified as a character vector, string scalar, or MATLAB date number that denotes the beginning of the date range to retrieve data.

Data Types: `double` | `char` | `string`

`enddate` — End date
character vector | string scalar | MATLAB date number

End date, specified as a character vector, string scalar, or MATLAB date number that denotes the end of the date range to retrieve data.

Data Types: `double` | `char` | `string`

`period` — Period
`'d'` | `'w'` | `'m'` | `'q'` | `'a'`

Period, specified as one of these values that denotes the time period for the historical data:

- `'d'` for daily values
- `'w'` for weekly values
- `'m'` for monthly values
- `'q'` for quarterly values
- `'a'` for annual values

Output Arguments

`d` — Historical data
matrix

Historical data, returned as a matrix with the numeric representation of the date in the first column and the value in the second column.

Version History

Introduced in R2007a

See Also

`close` | `get` | `isconnection` | `haver` | `info` | `nextinfo`

Topics

“Retrieve Historical Data Using Haver Analytics” on page 1-12

get

Retrieve properties from Haver Analytics connection objects

Syntax

```
V = get(c,'PropertyName')
V = get(c)
```

Arguments

`c`	Haver Analytics connection object created with `haver`.
`'PropertyName'`	A MATLAB character vector, string, cell array of character vectors, or string array containing property names. The property name is `Databasename`.

Description

`V = get(c,'PropertyName')` returns a MATLAB structure containing the value of the specified properties for the Haver Analytics connection object.

`V = get(c)` returns a MATLAB structure, where each field name is the name of a property of `c`. Each field contains the value of the property.

Examples

Establish a Haver Analytics connection.

```
c = haver('d:\work\haver\data\haverd.dat')
```

Retrieve the name of the Haver Analytics database.

```
V = get(c,'DatabaseName')

  V =

    DatabaseName: 'd:\work\haver\data\haverd.dat'
```

Version History

Introduced in R2007a

See Also

`close` | `fetch` | `isconnection` | `haver`

Topics

“Retrieve Historical Data Using Haver Analytics” on page 1-12

info

Retrieve information about Haver Analytics variables

Syntax

```
D = info(c,s)
```

Arguments

`c`	Haver Analytics connection object created with `haver`.
`s`	Haver Analytics variable.

Description

`D = info(c,s)` returns information about the Haver Analytics variable, `s`.

Examples

Retrieve Information About Variable

Using a Haver Analytics connection, retrieve information about a variable.

Establish a Haver Analytics connection `c`.

```
c = haver('d:\work\haver\data\haverd.dat');
```

Request information for the variable `'FFED2'`. The variable `d` is a structure with a field for each piece of information.

```
s = 'FFED2';
d = info(c,s)
```

```
d =

  struct with fields:

     VarName: 'FFED2'
   StartDate: '01-Jan-1991'
     EndDate: '31-Dec-1998'
   NumberObs: 2088
   Frequency: 'D'
 DateTimeMod: '02-Apr-2007 20:46:37'
   Magnitude: 0
DecPrecision: 2
     DifType: 1
     AggType: 'AVG'
    DataType: '%'
       Group: 'Z05'
      Source: 'FRB'
```

```
  Descriptor: 'Federal Funds [Effective] Rate (% p.a.)'
 ShortSource: 'History'
  LongSource: 'Historical Series'
```

Close the Haver Analytics connection.

```
close(c)
```

Retrieve Information About Variable as Table

Using a Haver Analytics connection, retrieve information about a variable. Return information as a table.

Establish a Haver Analytics connection `c`.

```
c = haver('d:\work\haver\data\haverd.dat');
```

Set the data return format to a table using the `DataReturnFormat` property of the `haver` object.

```
c.DataReturnFormat = 'table';
```

Request information for the variable `'ABQ'`. The variable `ABQ` provides bankruptcy filings in the US. `d` is a table with a variable for each piece of information.

```
s = 'ABQ';
d = info(c,s)
```

```
d =

  1×16 table

    VarName      StartDate         EndDate       NumberObs    Frequency         DateTimeMod
    _______    _____________    _____________    _________    _________    ____________________

     'ABQ'     '01-Jan-1980'    '01-Jan-2015'     141.00         'Q'       '28-Apr-2015 16:21:22
```

Close the Haver Analytics connection.

```
close(c)
```

Retrieve Information About Variable with Datetime

Using a Haver Analytics connection, retrieve information about a variable. Return dates as `datetime` arrays.

Establish a Haver Analytics connection `c`.

```
c = haver('d:\work\haver\data\haverd.dat');
```

Set the date and time format to a `datetime` array using the `DatetimeType` property of the `haver` object.

```
c.DatetimeType = 'datetime';
```

Request information for the variable `'ABQ'`. The variable `ABQ` provides bankruptcy filings in the US.

```
s = 'ABQ';
d = info(c,s)
```

```
d =

  struct with fields:

         VarName: 'ABQ'
       StartDate: 01-Jan-1980
         EndDate: 01-Jan-2015
       NumberObs: 141.00
       Frequency: 'Q'
     DateTimeMod: 28-Apr-2015 16:21:22
       Magnitude: 0
    DecPrecision: 0
         DifType: 0
         AggType: 'SUM'
        DataType: 'Units'
           Group: 'C13'
          Source: 'USC'
      Descriptor: 'Total Bankruptcy Filings, U.S. (Units)'
     ShortSource: 'USCOURTS'
      LongSource: 'Administrative Office of the U.S. Courts'
```

`d` is a structure with a field for each piece of information. The dates are `datetime` arrays.

Close the Haver Analytics connection.

```
close(c)
```

Version History

Introduced in R2007a

See Also

`close` | `get` | `isconnection` | `haver` | `nextinfo`

Topics

“Retrieve Historical Data Using Haver Analytics” on page 1-12

isconnection

Determine if connections to Haver Analytics data servers are valid

Syntax

```
X = isconnection(c)
```

Arguments

`c`	Haver Analytics connection object created with `haver`.

Description

`X = isconnection(c)` returns `X = 1` if the connection is a valid Haver Analytics connection, and `X = 0` otherwise.

Examples

Establish a Haver Analytics connection `c`:

```
c = HAVER('d:\work\haver\data\haverd.dat');
```

Verify that `c` is a valid Haver Analytics connection:

```
X = isconnection(c)
X = 1
```

Version History

Introduced in R2007a

See Also

`haver` | `close` | `fetch` | `get`

Topics

“Retrieve Historical Data Using Haver Analytics” on page 1-12

nextinfo

Retrieve information about next Haver Analytics variable

Syntax

```
D = nextinfo(c,s)
```

Arguments

c	Haver Analytics connection object created with the haver function.
s	Haver Analytics variable.

Description

`D = nextinfo(c,s)` returns information for the next Haver Analytics variable after the variable, `s`.

Examples

Retrieve Information About Next Variable

Establish a Haver Analytics connection.

```
c = haver('d:\work\haver\data\usecon.dat');
```

Request information for the variable following `'FFED'`. The variable `d` is a structure with a field for each piece of information.

```
s = 'FFED';
d = nextinfo(c,s)
```

```
d =

  struct with fields:

     VarName: 'FFED2'
   StartDate: '01-Jan-1991'
     EndDate: '31-Dec-1998'
   NumberObs: 2088
   Frequency: 'D'
 DateTimeMod: '02-Apr-2007 20:46:37'
   Magnitude: 0
DecPrecision: 2
     DifType: 1
     AggType: 'AVG'
    DataType: '%'
       Group: 'Z05'
      Source: 'FRB'
  Descriptor: 'Federal Funds [Effective] Rate (% p.a.)'
 ShortSource: 'History'
  LongSource: 'Historical Series'
```

Close the Haver Analytics connection.

```
close(c)
```

Retrieve Information About Next Variable as Table

Establish a Haver Analytics connection.

```
c = haver('d:\work\haver\data\usecon.dat');
```

Set the data return format to a table using the `DataReturnFormat` property of the `haver` object.

```
c.DataReturnFormat = 'table';
```

Request information for the variable following `'ABQ'`. The variable `d` is a table with a variable for each piece of information.

```
s = 'ABQ';
d = nextinfo(c,s)
```

```
d =

  1×16 table

    VarName       StartDate         EndDate       NumberObs    Frequency         DateTimeMod
    ________    _____________    _____________    _________    _________    ____________________

    'ACPIF1'    '01-Jan-1967'    '01-Apr-2015'     580.00         'M'       '22-May-2015 14:38:31
```

Close the Haver Analytics connection.

```
close(c)
```

Retrieve Information About Next Variable with Datetime

Establish a Haver Analytics connection.

```
c = haver('d:\work\haver\data\usecon.dat');
```

Set the date and time format to a `datetime` array using the `DatetimeType` property of the `haver` object.

```
c.DatetimeType = 'datetime';
```

Request information for the variable following `'ABQ'`.

```
s = 'ABQ';
d = nextinfo(c,s)
```

```
d =

  struct with fields:

         VarName: 'ACPIF1'
       StartDate: 01-Jan-1967
```

```
         EndDate: 01-Apr-2015
       NumberObs: 580.00
       Frequency: 'M'
    DateTimeMod: 22-May-2015 14:38:33
       Magnitude: 0
   DecPrecision: 2.00
        DifType: 1.00
        AggType: 'NA'
       DataType: '%'
          Group: 'P25'
         Source: 'STLF'
     Descriptor: 'Atlanta Fed Flexible CPI (SAAR, %chg)'
    ShortSource: 'FRBATL'
     LongSource: 'Federal Reserve Bank of Atlanta'
```

The variable `d` is a structure with a field for each piece of information. The dates are `datetime` arrays.

Close the Haver Analytics connection.

```
close(c)
```

Version History

Introduced in R2007a

See Also

`close` | `get` | `haver` | `info` | `isconnection`

Topics

“Retrieve Historical Data Using Haver Analytics” on page 1-12

havertool

Run Haver Analytics graphical user interface (GUI)

Syntax

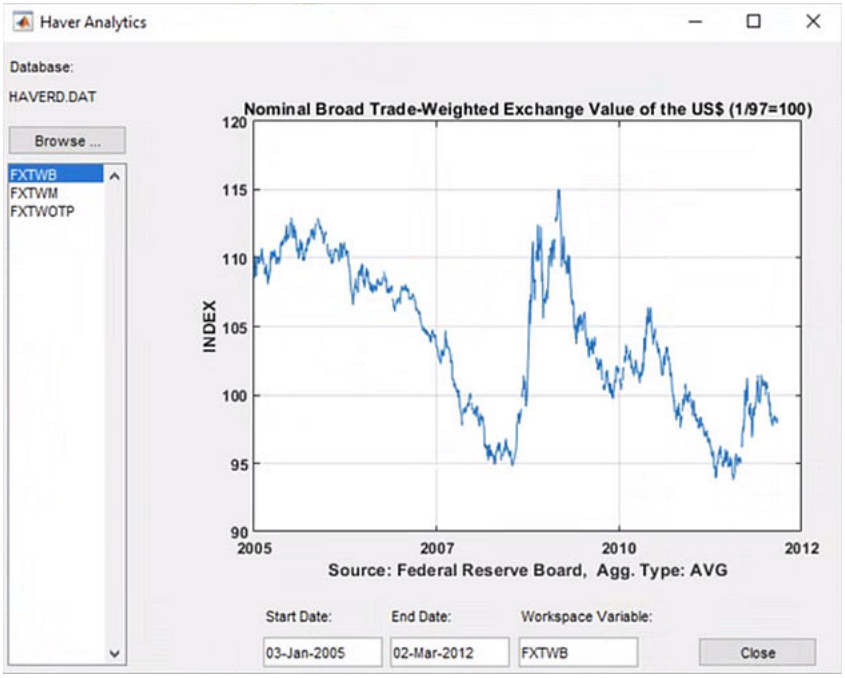
```
havertool(c)
```

Arguments

`c`	Haver Analytics connection object created with `haver`.

Description

`havertool(c)` runs the Haver Analytics graphical user interface (GUI). The GUI appears in the following figure.



The GUI fields and buttons are:

- **Database**: The currently selected Haver Analytics database.
- **Browse**: Allows you to browse for Haver Analytics databases, and populates the variable list with the variables in the database you specify.
- **Start Date**: The data start date of the selected variable.
- **End Date**: The data end date of the selected variable.
- **Workspace Variable**: The MATLAB variable to which `havertool` writes data for the currently selected Haver Analytics variable.

- **Close**: Closes all current connections and the Haver Analytics GUI.

Examples

Establish a Haver Analytics connection H:

```
c = haver('d:\work\haver\data\haverd.dat');
```

Open the graphical user interface (GUI) demonstration:

```
havertool(c)
```

Version History

Introduced in R2007a

See Also

haver

Topics

“Retrieve Historical Data Using Haver Analytics” on page 1-12

krg

Create Kissell Research Group transaction cost analysis object

Description

To begin a transaction cost analysis, use MATLAB to retrieve the encrypted market-impact parameters from the Kissell Research Group (KRG) FTP site. Then, use the `krg` function to create a `krg` object in which to store the encrypted data. After you create a `krg` object, you can use the object functions to estimate trading costs, optimize trading strategies for a single stock or a portfolio, and conduct back testing and stress testing. For details about market-impact parameters and data, consult the Kissell Research Group. For a simple example of estimating trading costs, see “Estimate Trading Costs for Collection of Stocks” on page 6-33.

Creation

Syntax

```
k = krg(midata)
k = krg(midata,midate)
k = krg(midata,midate,micode)
k = krg(midata,midate,micode,tradedaysinyear)
```

Description

`k = krg(midata)` creates a transaction cost analysis object and sets the MiData property.

`k = krg(midata,midate)` also selects a market-impact date.

`k = krg(midata,midate,micode)` also sets the MiCode property.

`k = krg(midata,midate,micode,tradedaysinyear)` also sets the TradeDaysInYear property.

Input Arguments

`midate` — Market-impact date
double | character vector | string | `datetime` array

Market-impact date, specified as a double, character vector, string, or `datetime` array. By default, the market-impact date is the current date. To decrypt market-impact parameters for a specific date, specify the date using this input argument. For details, consult the Kissell Research Group.

Example: `'yesterday'`

Data Types: `double` | `char` | `string` | `datetime`

Properties

`MiData` — KRG market-impact data
table

Market-impact data, specified as a table. This table contains the encrypted market-impact date, code, and parameters. Retrieve this data from the KRG FTP site ftp://ftp.kissellresearch.com using your user name and password. For details, consult the Kissell Research Group.

Example: `[276x12 table]`

Data Types: `table`

`MiDate` — KRG market-impact date
`datetime` array

Market-impact date, specified as a `datetime` array. By default, the market-impact date is the current date. To decrypt market-impact parameters for a specific date, specify the date using the `midate` input argument. For details, consult the Kissell Research Group.

The `krg` function sets this property using the `midate` input argument.

Example: `09-Sep-2015`

Data Types: `datetime`

`MiCode` — KRG market-impact code
numeric scalar

Market-impact code, specified as a numeric scalar. By default, the market-impact code is 1. To decrypt market-impact parameters for a specific market region, specify the code by setting this property using dot notation. For details, consult the Kissell Research Group.

Example: `1`

Data Types: `double`

`TradeDaysInYear` — Number of trading days in year
250 (default) | numeric scalar

Number of trading days in the year, specified as a numeric scalar.

Example: `251`

Data Types: `double`

Object Functions

costCurves	Estimate market-impact cost of order execution
iStar	Estimate instantaneous trading cost for order
liquidityFactor	Estimate and compare liquidation costs across stocks
marketImpact	Estimate price movement due to order or trade
portfolioCostCurves	Estimate market-impact cost of order execution for portfolio
priceAppreciation	Estimate trading cost due to natural price movement
timingRisk	Estimate uncertainty of market impact cost

Examples

Create Transaction Cost Analysis Object

First, retrieve market-impact data from KRG. Then, create a transaction cost analysis object and estimate trading costs for the current day.

Retrieve the market impact data from the KRG FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a KRG transaction cost analysis object `k`.

```
k = krg(miData)
```

```
k =

  krg with properties:

            MiData: [276x12 table]
            MiDate: 09-Sep-2015
            MiCode: 1.00
    TradeDaysInYear: 250.00
```

`k` has these properties:

- Market-impact data
- Market-impact date
- Market-impact code
- Number of trading days in the year

Load the example data `TradeData` from the file `KRGExampleData.mat`, which is included with Datafeed Toolbox.

```
load KRGExampleData.mat TradeData
```

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Estimate the instantaneous trading cost `itc` using `TradeData`.

```
itc = iStar(k,TradeData);
```

You can estimate other trading costs using the market activity for the current day. For details, see “Estimate Trading Costs for Collection of Stocks” on page 6-33.

Create Transaction Cost Analysis Object with Market-Impact Date

First, retrieve market-impact data from KRG. Then, create a transaction cost analysis object using a specific date and estimate trading costs for that date.

Retrieve the market impact data from the KRG FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a KRG transaction cost analysis object `k` with a specific market-impact date `midate`. Set the date to yesterday.

```
midate = 'yesterday';

k = krg(miData,midate)
```

```
k =

  krg with properties:

            MiData: [276x12 table]
            MiDate: 09-Sep-2015
            MiCode: 1.00
    TradeDaysInYear: 250.00
```

Load the example data `TradeData` from the file `KRGExampleData.mat`, which is included with Datafeed Toolbox.

```
load KRGExampleData.mat TradeData
```

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Estimate the instantaneous trading cost `itc` using `TradeData`.

```
itc = iStar(k,TradeData);
```

You can estimate other trading costs using the market activity for yesterday. For details, see “Estimate Trading Costs for Collection of Stocks” on page 6-33.

Create Transaction Cost Analysis Object with Market-Impact Code

First, retrieve market-impact data from the KRG. Then, create a transaction cost analysis object using a specific market-impact code, and estimate trading costs for a particular market region.

Retrieve the market impact data from the KRG FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a KRG transaction cost analysis object `k` with a specific market-impact code `micode`. Set the date to yesterday. Set the code to 1.

```
midate = 'yesterday';
micode = 1;
```

```
k = krg(miData,midate,micode)

k =

  krg with properties:

            MiData: [276x12 table]
            MiDate: 09-Sep-2015
            MiCode: 1.00
    TradeDaysInYear: 250.00
```

Load the example data `TradeData` from the file `KRGExampleData.mat`, which is included with Datafeed Toolbox.

```
load KRGExampleData.mat TradeData
```

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Estimate the instantaneous trading cost `itc` using `TradeData`.

```
itc = iStar(k,TradeData);
```

Using the market activity for yesterday, you can estimate trading costs for a particular market region. For details, see “Estimate Trading Costs for Collection of Stocks” on page 6-33.

Create Transaction Cost Analysis Object with Number of Trading Days

First, retrieve market-impact data from the KRG. Then, create a transaction cost analysis object using a specified number of trading days, and estimate trading costs for those trading days.

Retrieve the market impact data from the KRG FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a KRG transaction cost analysis object `k` with a specific number of trading days in the year `tradedays`. Set the number of trading days to 251. Enter `[]` for the market-impact date and code so that `krg` sets these input arguments to their default values.

```
tradedays = 251;

k = krg(miData,[],[],tradedays)

k =

  krg with properties:

            MiData: [276x12 table]
            MiDate: 09-Sep-2015
```

```
            MiCode: 1.00
    TradeDaysInYear: 251.00
```

Load the example data `TradeData` from the file `KRGExampleData.mat`, which is included with Datafeed Toolbox.

```
load KRGExampleData.mat TradeData
```

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Estimate the instantaneous trading cost `itc` using `TradeData`.

```
itc = iStar(k,TradeData);
```

Using the market activity for yesterday, you can estimate trading costs for a particular market region with 251 trading days in the year. For details, see “Estimate Trading Costs for Collection of Stocks” on page 6-33.

Modify Transaction Cost Analysis Object Property

First, retrieve market-impact data from the KRG. Then, create a transaction cost analysis object and set the market-impact date using the object properties.

Retrieve the market impact data from the KRG FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a KRG transaction cost analysis object `k` using `miData`.

```
k = krg(miData);
```

Modify the `MiDate` property to retrieve market-impact data from a different day.

```
k.MiDate = '05-Dec-2015'
```

```
k =

  krg with properties:

            MiData: [276x12 table]
            MiDate: '05-Dec-2015'
            MiCode: 1.00
    TradeDaysInYear: 251.00
```

You can estimate trading costs using the market activity for the specified day. For details, see “Estimate Trading Costs for Collection of Stocks” on page 6-33.

Tips

If the market-impact code does not exist in the market-impact data, this error appears.

```
The given region code does not match any records in the market impact data.
```

Version History

Introduced in R2016a

See Also

`iStar` | `marketImpact` | `priceAppreciation` | `timingRisk`

Topics

“Analyze Trading Execution Results” on page 6-2
“Estimate Portfolio Liquidation Costs” on page 6-20
“Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17
“Optimize Percentage of Volume Trading Strategy” on page 6-23
“Optimize Trade Time Trading Strategy” on page 6-26
“Optimize Trade Schedule Trading Strategy” on page 6-29

External Websites

ftp://ftp.kissellresearch.com

costCurves

Estimate market-impact cost of order execution

Syntax

```
cc = costCurves(k,trade,tradeQuantity,tqRange,tradeStrategy,tsRange)
```

Description

`cc = costCurves(k,trade,tradeQuantity,tqRange,tradeStrategy,tsRange)` returns the market-impact costs of order execution using:

- Kissell Research Group (KRG) transaction cost analysis object `k`
- Trade data `trade`
- Trade quantity `tradeQuantity` with a range of values `tqRange`
- Trade strategy `tradeStrategy` with a range of values `tsRange`

Examples

Estimate Market-Impact Cost for an Order

Retrieve the market impact data from the KRG FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`.

```
k = krg(miData);
```

Load the example data from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox.

```
load KRGExampleData
```

The variable `TradeData` appears in the MATLAB workspace.

`TradeData` contains these variables:

- Stock symbol
- Stock price
- Average daily volume

- Volatility

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Estimate market-impact costs with the trade quantity `'Size'` and strategy `'POV'`. Specify the trade quantity range with increments of 0.01 by starting from 0.01 and ending at one. Specify the trade strategy range with increments of 0.05 by starting from 0.05 and ending at 0.5.

```
cc = costCurves(k,TradeData,'Size',(0.01:0.01:1),'POV',(0.05:0.05:0.5));
```

Display the first three rows of market-impact cost data.

```
cc(1:3,:)
```

```
ans =

    Symbol    Size     Shares       Dollars      POV     TradeTime    Cost_BP    Cost_DollarsPerS
    ______    ____    _________    __________    ____    _________    _______    ________________

    'AAL'     0.01    114764.24    6251208.50    0.05    0.19         11.42      0.06
    'AAL'     0.01    114764.24    6251208.50    0.10    0.09         17.93      0.10
    'AAL'     0.01    114764.24    6251208.50    0.15    0.06         23.42      0.13
```

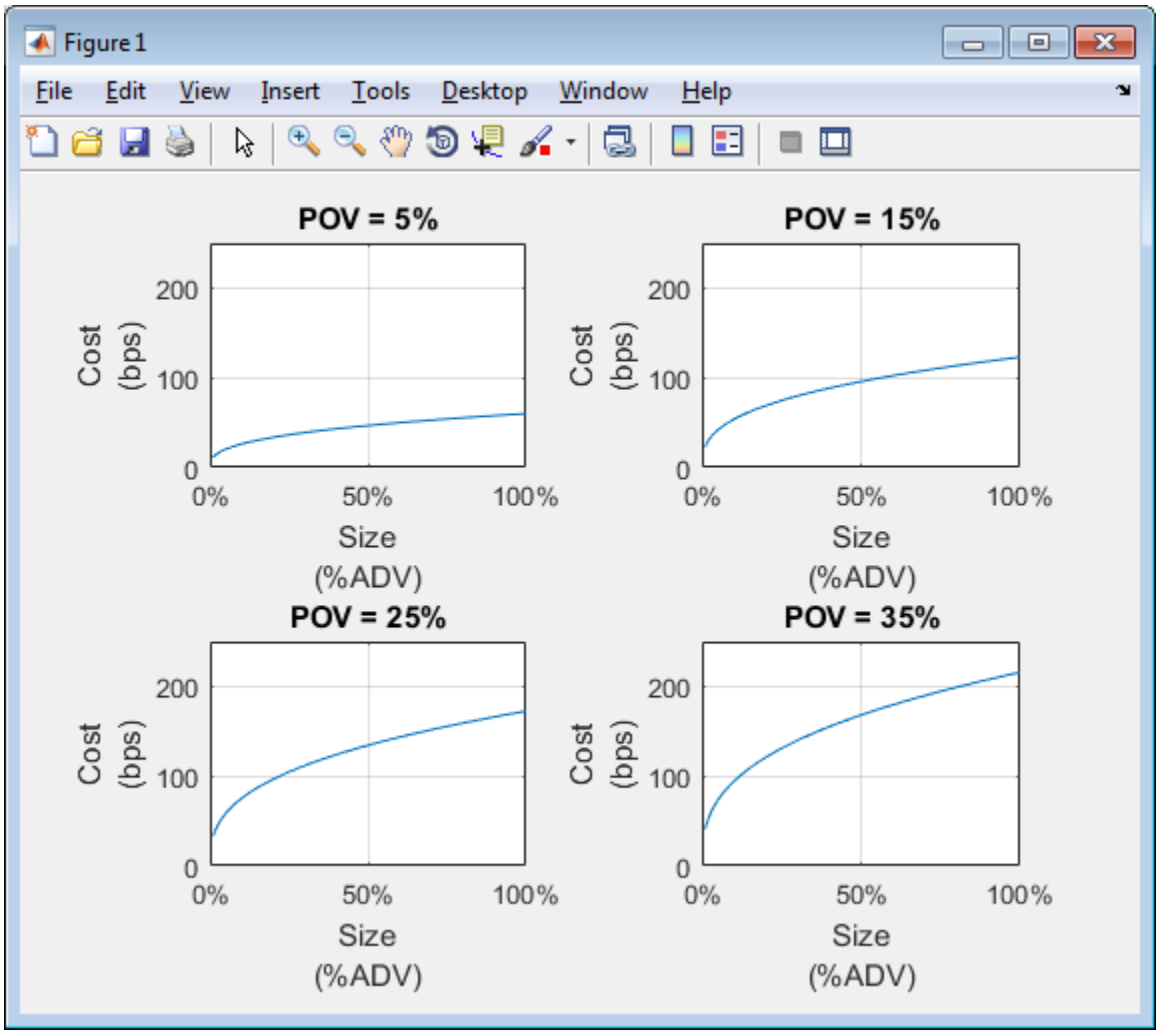
The market-impact cost data contains:

- Stock symbol
- Size
- Number of shares in the transaction
- Dollar amount of the transaction
- Percentage of volume to complete the transaction
- Trade time to complete the transaction in percentage of the day
- Market-impact cost in basis points
- Market-impact cost in dollars per share
- Market-impact cost in dollars

Display cost curves for the first stock for these percentage of volume rates: 5%, 15%, 25%, and 35%.

```
figure
subplot(2,2,1)
plot(cc.Size(1:10:1000)*100,cc.Cost_BP(1:10:1000))
grid on
axis([0 100 0 250])
xlabel({'Size','(%ADV)'})
ylabel({'Cost','(bps)'})
title('POV = 5%')
a = gca;
a.XAxis.TickLabelFormat = '%g%%';

subplot(2,2,2)
plot(cc.Size(3:10:1000)*100,cc.Cost_BP(3:10:1000))
grid on
axis([0 100 0 250])
xlabel({'Size','(%ADV)'})
ylabel({'Cost','(bps)'})
```

```
title('POV = 15%')
b = gca;
b.XAxis.TickLabelFormat = '%g%%';

subplot(2,2,3)
plot(cc.Size(5:10:1000)*100,cc.Cost_BP(5:10:1000))
grid on
axis([0 100 0 250])
xlabel({'Size','(%ADV)'})
ylabel({'Cost','(bps)'})
title('POV = 25%')
c = gca;
c.XAxis.TickLabelFormat = '%g%%';

subplot(2,2,4)
plot(cc.Size(7:10:1000)*100,cc.Cost_BP(7:10:1000))
grid on
axis([0 100 0 250])
xlabel({'Size','(%ADV)'})
ylabel({'Cost','(bps)'})
title('POV = 35%')
d = gca;
d.XAxis.TickLabelFormat = '%g%%';
```



This figure demonstrates how fast to trade a specific order size within a price level.

Input Arguments

k — Transaction cost analysis

KRG object

Transaction cost analysis, specified as a KRG object created using `krg`.

`trade` — Trade data

table | structure

Trade data that describes the stocks in the transaction, specified as a table or structure. `trade` must contain these variable or field names.

Variable or Field Name	Description
`Symbol`	Stock symbol
`Price`	Stock price

Variable or Field Name	Description
ADV	Average daily volume
Volatility	Volatility

Example: `trade = table({'XYZ'},100.00,860000,0.27,'VariableNames',{'Symbol' 'Price' 'ADV' 'Volatility'})`

Example: `trade = struct('Symbol','XYZ','Price',100.00,'ADV',860000,'Volatility',0.27)`

These examples do not represent real market data.

Data Types: `struct` | `table`

`tradeQuantity` — Trade quantity
`'Size'` | `'Shares'` | `'Dollars'`

Trade quantity, specified as one of these values.

Value	Trade Quantity Description
`'Size'`	Shares in the transaction, which is a percentage of average daily trading volume
`'Shares'`	Number of shares in the transaction
`'Dollars'`	Total value of the transaction

`tqRange` — Trade quantity range
vector

Trade quantity range, specified as a vector. `costCurves` uses these values with the trade strategy range values to estimate market-impact costs for different quantities and strategies.

Example: `'Size',(0.01:0.01:1)` specifies a trade quantity range with increments of 0.01 starting from 0.01 and ending at one

Data Types: `double`

`tradeStrategy` — Trade strategy
`'POV'` | `'TradeTime'`

Trade strategy, specified as one of these values.

Values	Trade Strategy Name
`'POV'`	Percentage of volume
`'TradeTime'`	Trade time in percentage of the day

`tsRange` — Trade strategy range
vector

Trade strategy range, specified as a vector. `costCurves` uses these values with the trade quantity range values to estimate market-impact costs for different quantities and strategies.

Example: `'POV',(0.05:0.05:0.5)` specifies a trade strategy range with increments of 0.05 starting from 0.05 and ending at 0.5

Data Types: `double`

Output Arguments

`cc` — Cost curves

table | structure

Cost curves, returned as a table or structure with these variable names or fields.

Variable or Field Name	Description
`Symbol`	Stock symbol
`Size`	Shares in a transaction in percentage of average daily trading volume
`Shares`	Number of shares in the transaction
`Dollars`	Dollar amount of the transaction
`POV`	Percentage of volume to complete the transaction
`TradeTime`	Trade time to complete the transaction in percentage of the day
`Cost_BP`	Market-impact cost of the transaction in basis points
`Cost_DollarsPerShare`	Market-impact cost of the transaction in dollars per share
`Cost_Dollars`	Market-impact cost of the transaction in dollars

Tips

- For details about the calculations, contact Kissell Research Group.

Version History

Introduced in R2016a

References

[1] Kissell, Robert. “A Practical Framework for Transaction Cost Analysis.” *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. “Algorithmic Trading Strategies.” Ph.D. Thesis. Fordham University, May 2006.

[3] Kissell, Robert. “TCA in the Investment Process: An Overview.” *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[4] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[5] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`krg` | `iStar` | `marketImpact` | `portfolioCostCurves` | `timingRisk`

Topics

“Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17

iStar

Estimate instantaneous trading cost for order

Syntax

```
itc = iStar(k,trade)
```

Description

`itc = iStar(k,trade)` returns the instantaneous trading cost of an order using the Kissell Research Group (KRG) transaction cost analysis object `k` and trade data `trade`. To estimate the instantaneous trading cost, `iStar` uses the I-Star trading cost model on page 15-845.

Examples

Estimate Instantaneous Trading Cost for Stocks

Retrieve the market impact data from the KRG FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`.

```
k = krg(miData);
```

Load the example data from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox.

```
load KRGExampleData
```

The variable `TradeData` appears in the MATLAB workspace.

`TradeData` contains these variables:

- Stock symbol
- Side
- Number of shares
- Size
- Stock price
- Average daily volume

- Volatility
- Percentage of volume

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Estimate instantaneous trading cost `itc` for each stock using the Kissell Research Group transaction cost analysis object `k`. Display the first three instantaneous trading costs.

```
itc = iStar(k,TradeData);

itc(1:3)

ans =

         33.48
        317.58
         62.94
```

Instantaneous trading costs display in basis points.

Input Arguments

k — Transaction cost analysis

KRG object

Transaction cost analysis, specified as a KRG object created using `krg`.

trade — Trade data

table | structure

Trade data that describes the stocks in the transaction, specified as a table or structure. `trade` must contain these variable or field names.

Variable or Field Name	Description
`Symbol`	Stock symbol
`Side`	Buy or sell side
`Shares`	Number of shares in the transaction
`Size`	Shares in the transaction, which is a percentage of average daily trading volume
`Price`	Stock price
`ADV`	Average daily volume
`Volatility`	Volatility
`POV`	Percentage of volume

The trading cost varies with the trade strategy. `iStar` determines the trade strategy using these variables in this order:

1. Percentage of volume
2. Trade time
3. Trade schedule

To change the trade strategy from percentage of volume to trade time, remove the variable `POV` in the table and add the variable `TradeTime` with trade time data. To use the trade schedule strategy, remove the variable `TradeTime` and add the `TradeSchedule` and `VolumeProfile` variables.

If you specify size in the trade data, `iStar` uses the `Size` variable. Otherwise, `iStar` uses the variables `ADV` and `Shares` to determine the size.

For example, to create trade data as a table, enter:

```
trade = table({'XYZ'},{'Buy'},9300,0.06,29.68,860000,0.27,0.17,...
    'VariableNames',{'Symbol' 'Side' 'Shares' 'Size' 'Price' ...
    'ADV' 'Volatility' 'POV'})
```

To create trade data as a structure, enter:

```
trade.Symbol = {'XYZ'};
trade.Side = {'Buy'};
trade.Shares = 9300;
trade.Size = 0.06;
trade.Price = 29.68;
trade.ADV = 860000;
trade.Volatility = 0.27;
trade.POV = 0.17;
```

These examples do not represent real market data.

Data Types: `struct` | `table`

Output Arguments

`itc` — Instantaneous trading cost

vector

Instantaneous trading cost, returned as a vector. The vector values correspond to the instantaneous trading cost in basis points for each stock in `trade`.

More About

I-Star Trading Cost Model

The I-Star trading cost model (I-Star) estimates the instantaneous cost of an order. If a market participant immediately releases the entire order to the market for execution, they incur this cost. This cost also refers to the market participant cost accounting for 100% of the market volume over the execution period.

The I-Star model is

$$I^* = a_1 \cdot \left(\frac{Shares}{ADV}\right)^{a_2} \cdot \sigma^{a_3}.$$

Shares are the number of shares to trade. *ADV* is the average daily volume of the stock. σ is the price volatility. a_1, a_2, and a_3 are the model parameters.

Model Parameter	Description
a_1	Price sensitivity to order flow
a_2	Order size shape
a_3	Volatility shape

The general I-Star model that includes stock-specific factors is

$$I^* = a_1 \cdot \left(\frac{Shares}{ADV}\right)^{a_2} \cdot \sigma^{a_3} \cdot Price^{a_5} \cdot X_k^{a_k}.$$

Price is the stock price. a_5 is the price shape model parameter. X_k is the stock-specific factor such as market capitalization, beta, P/E ratio, and Debt/Equity ratio. This formulation can include multiple stock-specific factors. a_k is the corresponding shape parameter for the stock-specific factor X_k.

Tips

- For details about the formula and calculations, contact the Kissell Research Group.

Version History

Introduced in R2016a

References

[1] Kissell, Robert. “A Practical Framework for Transaction Cost Analysis.” *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. “Algorithmic Trading Strategies.” Ph.D. Thesis. Fordham University, May 2006.

[3] Kissell, Robert. “Creating Dynamic Pre-Trade Models: Beyond the Black Box.” *Journal of Trading*. Vol. 6, Number 4, Fall 2011, pp. 8–15.

[4] Kissell, Robert. “TCA in the Investment Process: An Overview.” *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[5] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[6] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`krg` | `marketImpact` | `priceAppreciation` | `timingRisk` | `liquidityFactor`

Topics

“Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17

liquidityFactor

Estimate and compare liquidation costs across stocks

Syntax

```
lf = liquidityFactor(k,trade)
```

Description

`lf = liquidityFactor(k,trade)` returns the ratio of liquidation costs due to liquidity demand by stock for an equal investment value, or liquidity factor on page 15-849. `liquidityFactor` uses the Kissell Research Group (KRG) transaction cost analysis object `k` and trade data `trade`.

Examples

Determine Liquidity Factor for Stocks

Retrieve the market impact data from the KRG FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`.

```
k = krg(miData);
```

Load the example data from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox.

```
load KRGExampleData
```

The variable `TradeData` appears in the MATLAB workspace.

`TradeData` contains these variables:

- Stock symbol
- Stock price
- Average daily volume
- Volatility

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Determine liquidity factor `lf` for each stock using the Kissell Research Group transaction cost analysis object `k`. Display the first three liquidity factor values.

```
lf = liquidityFactor(k,TradeData);

lf(1:3)

ans =

          0.30
          2.37
          0.35
```

`lf` returns the ratios for stock comparison due to liquidity demands.

Input Arguments

k — Transaction cost analysis

KRG object

Transaction cost analysis, specified as a KRG object created using `krg`.

trade — Trade data

table | structure

Trade data that describes the stocks in the transaction, specified as a table or structure. `trade` must contain these variable or field names.

Variable or Field Name	Description
`Symbol`	Stock symbol
`Price`	Stock price
`ADV`	Average daily volume
`Volatility`	Volatility

Example: `trade = table({'XYZ'},100.00,860000,0.27,'VariableNames',{'Symbol' 'Price' 'ADV' 'Volatility'})`

Example: `trade = struct('Symbol','XYZ','Price',100.00,'ADV',860000,'Volatility',0.27)`

These examples do not represent real market data.

Data Types: `struct` | `table`

Output Arguments

lf — Liquidity factor

vector

Liquidity factor, returned as a vector. The vector values are ratios that compare the liquidation costs due to liquidity demands across stocks in `trade` for the dollar value and execution strategy.

More About

Liquidity Factor

The Liquidity Factor (LF) is a stock-specific measure of price sensitivity to investment dollars.

LF provides investors with a fair and consistent comparison of expected liquidation costs across stocks. LF incorporates stock-specific information to determine its sensitivity to order flow and investment dollars. The LF metric shows the ratio of liquidation costs due to liquidity demand by stock for an equal investment value in each stock. Market impact relies on the order size or shares traded which vary from order to order. LF provides an apples-to-apples comparison across financial instruments. Consider a stock I that has an LF = 0.10 and a stock II that has an LF = 0.20. Stock II is twice as expensive to transact for an equal dollar value. An investor buys or sells $1 million dollars of stock in stock I and stock II utilizing the same execution strategy. The cost of stock II is twice as large as stock I. The LF metric incorporates stock liquidity, volatility, and price to determine the LF trading cost parameter.

The LF model is

$$\mathrm{LF} = a_1 \cdot \left(\frac{1}{ADV}\right)^{a_2} \cdot \sigma^{a_3} \cdot \left(\frac{1}{Price}\right)^{a_2} \cdot Price^{a_5}.$$

σ is price volatility. *ADV* is the average daily volume of the stock. *Price* is the current stock price in local currency. a_1, a_2, a_3, and a_5 are the model parameters.

Model Parameter	Description
a_1	Price sensitivity to order flow
a_2	Order size shape
a_3	Volatility shape
a_5	Price shape

Tips

- For details about the formula and calculations, contact the Kissell Research Group.
- You can expand the LF model to include a stock-specific factor such as market capitalization, beta, P/E ratio, and Debt/Equity ratio. In this case, X_k denotes the stock-specific factor and a_k denotes the corresponding shape parameter. For details about implementing an expanded LF model, contact the Kissell Research Group.

Version History

Introduced in R2016a

References

[1] Kissell, Robert. “A Practical Framework for Transaction Cost Analysis.” *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. “Algorithmic Trading Strategies.” Ph.D. Thesis. Fordham University, May 2006.

[3] Kissell, Robert. “TCA in the Investment Process: An Overview.” *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[4] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[5] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`krg` | `iStar` | `marketImpact` | `priceAppreciation` | `timingRisk`

Topics

“Estimate Portfolio Liquidation Costs” on page 6-20

marketImpact

Estimate price movement due to order or trade

Syntax

```
mi = marketImpact(k,trade)
```

Description

`mi = marketImpact(k,trade)` returns the market impact on page 15-853 cost for stocks using the Kissell Research Group (KRG) transaction cost analysis object `k` and trade data `trade`.

Examples

Estimates Market-Impact Costs

Retrieve the market impact data from the KRG FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`.

```
k = krg(miData);
```

Load the example data from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox.

```
load KRGExampleData
```

The variable `TradeData` appears in the MATLAB workspace.

`TradeData` contains these variables:

- Stock symbol
- Side
- Number of shares
- Size
- Stock price
- Average daily volume
- Volatility

- Percentage of volume

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Estimates market-impact cost `mi` for each stock using the Kissell Research Group transaction cost analysis object `k`. Display the first three market-impact costs.

```
mi = marketImpact(k,TradeData);

mi(1:3)

ans =

          0.51
         96.86
         10.72
```

Market-impact costs display in basis points.

Input Arguments

k — Transaction cost analysis

KRG object

Transaction cost analysis, specified as a KRG object created using `krg`.

trade — Trade data

table | structure

Trade data that describes the stocks in the transaction, specified as a table or structure. `trade` must contain these variable or field names.

Variable or Field Name	Description
`Symbol`	Stock symbol
`Side`	Buy or sell side
`Shares`	Number of shares in the transaction
`Size`	Shares in the transaction, which is a percentage of average daily trading volume
`Price`	Stock price
`ADV`	Average daily volume
`Volatility`	Volatility
`POV`	Percentage of volume

The trading cost varies with the trade strategy. `marketImpact` determines the trade strategy using these variables in this order:

1. Percentage of volume
2. Trade time
3. Trade schedule

To change the trade strategy from percentage of volume to trade time, remove the variable `POV` in the table and add the variable `TradeTime` with trade time data. To use the trade schedule strategy, remove the variable `TradeTime` and add the `TradeSchedule` and `VolumeProfile` variables.

If you specify size in the trade data, `marketImpact` uses the `Size` variable. Otherwise, `marketImpact` uses the variables `ADV` and `Shares` to determine the size.

For example, to create trade data as a table, enter:

```
trade = table({'XYZ'},{'Buy'},9300,0.06,29.68,860000,0.27,0.17,...
    'VariableNames',{'Symbol' 'Side' 'Shares' 'Size' 'Price' ...
    'ADV' 'Volatility' 'POV'})
```

To create trade data as a structure, enter:

```
trade.Symbol = {'XYZ'};
trade.Side = {'Buy'};
trade.Shares = 9300;
trade.Size = 0.06;
trade.Price = 29.68;
trade.ADV = 860000;
trade.Volatility = 0.27;
trade.POV = 0.17;
```

These examples do not represent real market data.

Data Types: `struct` | `table`

Output Arguments

`mi` — Market-impact cost

vector

Market-impact cost, returned as a vector. The vector values correspond to the market-impact costs in basis points for each stock in `trade`.

More About

Market Impact

Market impact (MI) estimates the price movement in a stock caused by a particular trade or order.

Market-impact cost always causes adverse price movement. Buy orders push the stock price higher and sell orders push the stock price lower. Market-impact cost occurs for two reasons: liquidity demands of the traders or investor and the information content of the order. The liquidity demand of a buy order requires the buyer to provide the market a premium to attract additional sells into the market. The liquidity demand of a sell order causes the seller to offer the stock at a discount to attract additional buys into the market. The information content of the trade typically signals to the market that the stock is under- or overvalued. Buy orders tend to signal to the market that the stock is undervalued thus causing an increase in price to correct for the mispricing. Sell orders tend to signal to the market that the stock is overvalued thus causing a decrease in price to correct for the mispricing. Market-impact cost depends on order size, volatility, company characteristics, and prevailing market conditions over the trading horizon such as liquidity and intraday trading patterns.

MI for an order that executes instantaneously is equal to the I-Star trading cost model (I-Star). For details about I-Star, see `iStar`. When MI equals I-Star, the trading costs are high and prices move

adversely. Therefore, investors trade passively to reduce their cost. Thus, they slice the order and trade over time such as minutes, hours, or possibly days. `marketImpact` incorporates the trade strategy of the investors into the cost calculation.

The MI model is

$$\mathrm{MI} = b_1 \cdot I^* \cdot POV^{a_4} + (1 - b_1) \cdot I^* .$$

I^* is I-Star. *POV* is the percentage of market volume, or participation fraction, of the order. a_4 and b_1 are the model parameters.

Model Parameter	Description
a_4	Percentage of volume rate shape
b_1	Percentage of temporary market impact. Temporary impact is dependent upon the trading strategy. Temporary impact occurs because of the liquidity demands of the investor.
$1 - b_1$	Percentage of permanent market impact. Permanent impact is the unavoidable impact cost. The order does not control the permanent impact. Permanent impact occurs because of the information content of the trade.

Tips

- For details about the formula and calculations, contact the Kissell Research Group.

Version History

Introduced in R2016a

References

[1] Kissell, Robert. “A Practical Framework for Transaction Cost Analysis.” *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. “Algorithmic Trading Strategies.” Ph.D. Thesis. Fordham University, May 2006.

[3] Kissell, Robert. “Creating Dynamic Pre-Trade Models: Beyond the Black Box.” *Journal of Trading*. Vol. 6, Number 4, Fall 2011, pp. 8–15.

[4] Kissell, Robert. “TCA in the Investment Process: An Overview.” *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[5] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[6] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`krg` | `iStar` | `priceAppreciation` | `timingRisk` | `liquidityFactor`

Topics

“Analyze Trading Execution Results” on page 6-2
“Estimate Portfolio Liquidation Costs” on page 6-20
“Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17
“Optimize Percentage of Volume Trading Strategy” on page 6-23
“Optimize Trade Time Trading Strategy” on page 6-26
“Optimize Trade Schedule Trading Strategy” on page 6-29

portfolioCostCurves

Estimate market-impact cost of order execution for portfolio

Syntax

```
pcc = portfolioCostCurves(k,portfolio,tradeQuantity,tqRange,tradeStrategy,
tsRange)
```

Description

`pcc = portfolioCostCurves(k,portfolio,tradeQuantity,tqRange,tradeStrategy, tsRange)` returns the market-impact cost of order execution for a portfolio using:

- Kissell Research Group (KRG) transaction cost analysis object `k`
- Portfolio data `portfolio`
- Trade quantity `tradeQuantity` with a range of values `tqRange`
- Trade strategy `tradeStrategy` with a range of values `tsRange`

Examples

Estimate Market-Impact Cost for a Portfolio Order

Retrieve the market impact data from the KRG FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`.

```
k = krg(miData);
```

Load the example portfolio data from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox.

```
load KRGExampleData
```

The variable `PortfolioData` appears in the MATLAB workspace.

`PortfolioData` contains these variables:

- Stock symbol
- Local price

- Price in a different currency if applicable
- Average daily volume
- Volatility
- Number of shares

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Estimate market-impact cost for an order execution on a portfolio of assets. Specify the trade quantity as `DollarValue`. Specify the trade quantity range `tqRange` with increments of $10,000,000. Start with a total portfolio value of $100,000,000 and end with $500,000,000. Set the percentage of volume trading strategy `POV`. Specify the trade strategy range `tsRange` with increments of 10% by starting with a percentage of volume of 10% and ending with 40%.

```
tqRange = (100000000:10000000:500000000);
tsRange = (0.10:0.10:0.40);

pcc = portfolioCostCurves(k,PortfolioData,'DollarValue',tqRange,...
'POV',tsRange);
```

Display the first three rows of market-impact cost data.

```
pcc(1:3,:)
```

```
ans =

    Size      Shares       TradeValue     AbsTradeValue     POV     TradeTime     Cost_bp     Cost_D
    ____    __________    _____________    _____________    ____    _________    _______    ______

    0.02    5612057.03    100000000.00     328737579.09     0.10    0.18         38.74      0.07
    0.02    5612057.03    100000000.00     328737579.09     0.20    0.08         61.18      0.11
    0.02    5612057.03    100000000.00     328737579.09     0.30    0.05         80.07      0.14
```

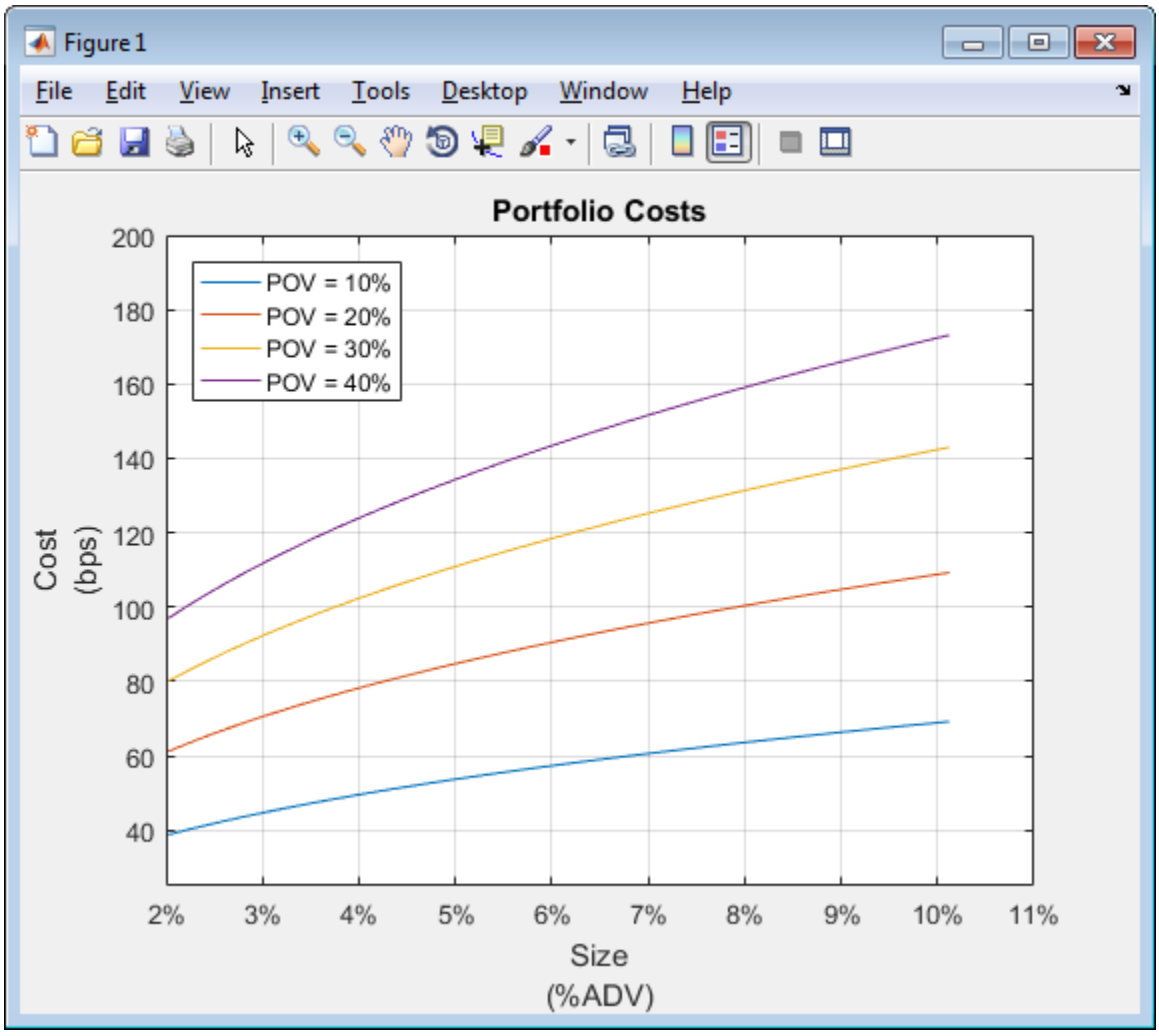
The market-impact cost data contains:

- Average trade size across all stocks in the portfolio
- Number of shares in the transaction
- Sum of traded value across all stocks in the portfolio
- Sum of absolute value of the trade value across all stocks in the portfolio
- Average execution percentage of volume to complete the number of shares
- Average trade time in percentage of the day to complete the number of shares
- Market-impact cost in basis points of local price
- Market-impact cost in dollars per share
- Market-impact cost in total dollar value

Display portfolio cost curves for percentage of volume rates: 10%, 20%, 30%, and 40%.

```
figure
size10 = pcc.Size(1:4:end)*100;
size20 = pcc.Size(2:4:end)*100;
size30 = pcc.Size(3:4:end)*100;
size40 = pcc.Size(4:4:end)*100;
cost10 = pcc.Cost_bp(1:4:end);
cost20 = pcc.Cost_bp(2:4:end);
```

```
cost30 = pcc.Cost_bp(3:4:end);
cost40 = pcc.Cost_bp(4:4:end);
plot(size10,cost10,size20,cost20,size30,cost30,size40,cost40)
grid on
axis([2 11 25 200])
xlabel({'Size','(%ADV)'})
ylabel({'Cost','(bps)'})
legend('POV = 10%','POV = 20%','POV = 30%','POV = 40%',...
'Location','northwest')
title('Portfolio Costs')
a = gca;
a.XAxis.TickLabelFormat = '%g%%';
```



This figure demonstrates using portfolio costs to construct the portfolio and manage portfolio contents. By analyzing portfolio costs, you can determine the optimal portfolio size.

Input Arguments

k — Transaction cost analysis

KRG object

Transaction cost analysis, specified as a KRG object created using `krg`.

`portfolio` — Portfolio data

table | structure

Portfolio data that describes the stocks in the portfolio, specified as a table or structure. `portfolio` must contain these variable or field names.

Variable or Field Name	Description
`Symbol`	Stock symbol.
`Price_Local`	Local price.
`Price_Currency`	Price, specified as the stock price with a different currency if the stock trades outside the United States. If the stock trades in the United States, the value equals the local price.
`ADV`	Average daily volume.
`Volatility`	Volatility.
`Shares`	Number of shares.

The number of symbols in the portfolio data must match the number of values for each market-impact parameter in the `miData` property of `k`. For details about the market-impact parameters, contact the Kissell Research Group.

Example: `portfolio = table({'XYZ'},100.00,100.00,860000,0.27,550,'VariableNames',{'Symbol' 'Price_Local' 'Price_Currency' 'ADV' 'Volatility' 'Shares'})`

Example: `portfolio = struct('Symbol','XYZ','Price_Local',100.00,'Price_Currency',100.00,'ADV',860000,'Volatility',0.27,'Shares',550)`

These examples do not represent real market data.

Data Types: `struct` | `table`

`tradeQuantity` — Trade quantity

`'DollarValue'` | `'PercentValue'`

Trade quantity, specified as one of these values.

Value	Trade Quantity Description
`'DollarValue'`	Total dollar value of the portfolio
`'PercentValue'`	Percentage of the total dollar value of the portfolio

`tqRange` — Trade quantity range

vector

Trade quantity range, specified as a vector. `portfolioCostCurves` uses these values with the trade strategy range values to estimate market-impact costs for different quantities and strategies.

Example: `'Size',(0.01:0.01:1)` specifies a trade quantity range with increments of 0.01 starting from 0.01 and ending at one

Data Types: `double`

`tradeStrategy` — Trade strategy
`'POV'` | `'TradeTime'`

Trade strategy, specified as one of these values.

Values	Trade Strategy Name
`'POV'`	Percentage of volume
`'TradeTime'`	Trade time in percentage of the day

`tsRange` — Trade strategy range
vector

Trade strategy range, specified as a vector. `portfolioCostCurves` uses these values with the trade quantity range values to estimate market-impact costs for different quantities and strategies.

Example: `'POV',(0.05:0.05:0.5)` specifies a trade strategy range with increments of 0.05 starting from 0.05 and ending at 0.5

Data Types: `double`

Output Arguments

`pcc` — Portfolio cost curves
table | structure

Portfolio cost curves, returned as a table or structure with these variable names or fields.

Variable or Field Name	Description
`Size`	Average trade size across all stocks in the portfolio.
`Shares`	Number of shares in the transaction.
`TradeValue`	Trade value, or the total dollar value of the stock position in the portfolio adjusted for side. Long/Buy positions have a positive trade value and Short/Sell positions have a negative trade value.
`AbsTradeValue`	Sum of absolute value of the trade value across all stocks in the portfolio.
`POV`	Average execution percentage of volume to complete the number of shares.
`TradeTime`	Average trade time in percentage of the day to complete the number of shares.
`Cost_bp`	Market-impact cost in basis points of local price.
`Cost_DollarsPerShare`	Market-impact cost in dollars per share.
`Cost_Dollars`	Market-impact cost in total dollar value.

Tips

- To test multiple portfolio transactions, you can use different ranges. You can change the percentage of shares in the transaction or use a different trade strategy. For details, see “Input Arguments” on page 15-858.
- For details about the calculations, contact Kissell Research Group.

Version History

Introduced in R2016a

References

[1] Kissell, Robert. “A Practical Framework for Transaction Cost Analysis.” *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. “Algorithmic Trading Strategies.” Ph.D. Thesis. Fordham University, May 2006.

[3] Kissell, Robert. “TCA in the Investment Process: An Overview.” *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[4] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[5] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`krg` | `costCurves` | `iStar` | `marketImpact` | `timingRisk`

Topics

“Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17

priceAppreciation

Estimate trading cost due to natural price movement

Syntax

```
alpha = priceAppreciation(k,trade)
```

Description

`alpha = priceAppreciation(k,trade)` returns the trading cost due to the natural price movement of a stock, or price appreciation on page 15-864. `priceAppreciation` uses the Kissell Research Group (KRG) transaction cost object `k` and trade data `trade`.

Examples

Estimate Alpha

Retrieve the market impact data from the KRG FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`.

```
k = krg(miData);
```

Load the example data from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox.

```
load KRGExampleData
```

The variable `TradeData` appears in the MATLAB workspace.

`TradeData` contains these variables:

- Shares in the transaction, which is a percentage of average daily trading volume
- Number of shares
- Average daily volume
- Percentage of volume
- Trade time in percentage of the day
- Volatility

- Stock price
- Alpha estimate

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Estimate alpha using the Kissell Research Group transaction cost analysis object `k`. Display the first three alphas.

```
alpha = priceAppreciation(k,TradeData);

alpha(1:3)

ans =

         -9.49
          8.47
          0.93
```

Alphas display in basis points.

Input Arguments

k — Transaction cost analysis

KRG object

Transaction cost analysis, specified as a KRG object created using `krg`.

trade — Trade data

table | structure

Trade data that describes the stocks in the transaction, specified as a table or structure. `trade` must contain these variable or field names.

Variable or Field Name	Description
`Size`	Shares in the transaction, which is a percentage of average daily trading volume
`Shares`	Number of shares
`ADV`	Average daily volume
`POV`	Percentage of volume
`TradeTime`	Trade time in percentage of the day
`Volatility`	Volatility
`Price`	Stock price
`Alpha_bp`	Alpha estimate in basis points

The trading cost varies with the trade strategy. `priceAppreciation` determines the trade strategy using these variables in this order:

1. Percentage of volume
2. Trade time
3. Trade schedule

To change the trade strategy from percentage of volume to trade time, remove the variable `POV` in the table and add the variable `TradeTime` with trade time data. To use the trade schedule strategy, remove the variable `TradeTime` and add the `TradeSchedule` and `VolumeProfile` variables.

If you specify size in the trade data, `priceAppreciation` uses the `Size` variable. Otherwise, `priceAppreciation` uses the variables `ADV` and `Shares` to determine the size.

Example: `trade = table(0.01,9300,860000,0.17,0.40,0.27,29.68,3,'VariableNames', {'Size' 'Shares' 'ADV' 'POV' 'TradeTime' 'Volatility' 'Price' 'Alpha_bp'})`

Example: `trade = struct('Size',0.01,'Shares',9300,'ADV',860000,'POV',0.17,'TradeTime',0.40,'Volatility',0.27,'Price',29.68,'Alpha_bp',3)`

These examples do not represent real market data.

Data Types: `struct` | `table`

Output Arguments

`alpha` — Alpha

vector

Alpha, returned as a vector. The units of alpha, or the natural price movement of the stock, are basis points.

More About

Price Appreciation

Price appreciation (PA) estimates the trading cost due to the natural price movement of a stock.

The natural price movement commonly refers to expected return, alpha, price trend, drift, or momentum. This movement represents how the stock moves in a market without any uncertainty. PA represents the trading cost due to the underlying trading strategy. For example, buying passively in a rising market or selling passively in a falling market causes the fund to incur higher costs due to market movement. Conversely, buying in a falling market or selling in a rising market causes the fund to incur lower costs due to transacting at the better prices. PA is based on the alpha estimate you specify in the trade data. Funds and managers heavily guard their alpha estimates and expected returns. These expectations are highly proprietary and valued. This function lets you input alpha estimates directly into the model running on your desktop that prevents information leakage.

The PA model is represented as a linear trend. The PA model is

$$\mathrm{PA} = 0.5 \cdot Alpha\_bp \cdot \left(\frac{Shares}{ADV}\right) \cdot \left(\frac{1 - POV}{POV}\right).$$

Shares are the number of shares to trade. *ADV* is the average daily volume of a stock. *POV* is the percent of market volume, or participation fraction, for the order. *Alpha_bp* is the alpha estimate for the day in basis points. A positive value for the alpha estimate indicates adverse price movement for the order. A negative value for the alpha estimate indicates favorable price movement.

Tips

- For details about the formula and calculations, contact the Kissell Research Group.

Version History

Introduced in R2016a

References

[1] Kissell, Robert. “A Practical Framework for Transaction Cost Analysis.” *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. “Algorithmic Trading Strategies.” Ph.D. Thesis. Fordham University, May 2006.

[3] Kissell, Robert. “TCA in the Investment Process: An Overview.” *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[4] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[5] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`krg` | `iStar` | `marketImpact` | `timingRisk` | `liquidityFactor`

Topics

“Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17
“Optimize Percentage of Volume Trading Strategy” on page 6-23
“Optimize Trade Time Trading Strategy” on page 6-26
“Optimize Trade Schedule Trading Strategy” on page 6-29

timingRisk

Estimate uncertainty of market impact cost

Syntax

```
tr = timingRisk(k,trade)
```

Description

`tr = timingRisk(k,trade)` returns the uncertainty of the market impact cost estimate, or timing risk on page 15-868. `timingRisk` uses the Kissell Research Group (KRG) transaction cost analysis object `k` and trade data `trade`.

Examples

Estimate Timing Risk for Stocks

Retrieve the market impact data from the KRG FTP site. Connect to the FTP site using the `ftp` function with a user name and password. Navigate to the `MI_Parameters` folder and retrieve the market impact data in the `MI_Encrypted_Parameters.csv` file. `miData` contains the encrypted market impact date, code, and parameters.

```
f = ftp('ftp.kissellresearch.com','username','pwd');
mget(f,'MI_Encrypted_Parameters.csv');

miData = readtable('MI_Encrypted_Parameters.csv','delimiter', ...
    ',','ReadRowNames',false,'ReadVariableNames',true);
```

Create a Kissell Research Group transaction cost analysis object `k`.

```
k = krg(miData);
```

Load the example data from the file `KRGExampleData.mat`, which is included with the Datafeed Toolbox.

```
load KRGExampleData
```

The variable `TradeData` appears in the MATLAB workspace.

`TradeData` contains these variables:

- Stock symbol
- Side
- Number of shares
- Size
- Stock price
- Average daily volume

- Volatility
- Percentage of volume

For a description of the example data, see “Kissell Research Group Data Sets” on page 6-7.

Estimate timing risk `tr` for each stock using the Kissell Research Group transaction cost analysis object `k`. Display the first three timing risk values.

```
tr = timingRisk(k,TradeData);

tr(1:3)

ans =

        159.05
        242.37
         62.88
```

Timing risk trading costs display in basis points.

Input Arguments

k — Transaction cost analysis

KRG object

Transaction cost analysis, specified as a KRG object created using `krg`.

trade — Trade data

table | structure

Trade data that describes the stocks in the transaction, specified as a table or structure. `trade` must contain these variable or field names.

Variable or Field Name	Description
`Symbol`	Stock symbol
`Side`	Buy or sell side
`Shares`	Number of shares in the transaction
`Size`	Shares in the transaction, which is a percentage of average daily trading volume
`Price`	Stock price
`ADV`	Average daily volume
`Volatility`	Volatility
`POV`	Percentage of volume

The trading cost varies with the trade strategy. `timingRisk` determines the trade strategy using these variables in this order:

1. Percentage of volume
2. Trade time
3. Trade schedule

To change the trade strategy from percentage of volume to trade time, remove the variable `POV` in the table and add the variable `TradeTime` with trade time data. To use the trade schedule strategy, remove the variable `TradeTime` and add the `TradeSchedule` and `VolumeProfile` variables.

If you specify size in the trade data, `timingRisk` uses the `Size` variable. Otherwise, `timingRisk` uses the variables `ADV` and `Shares` to determine the size.

For example, to create trade data as a table, enter:

```
trade = table({'XYZ'},{'Buy'},9300,0.06,29.68,860000,0.27,0.17,...
    'VariableNames',{'Symbol' 'Side' 'Shares' 'Size' 'Price' ...
    'ADV' 'Volatility' 'POV'})
```

To create trade data as a structure, enter:

```
trade.Symbol = {'XYZ'};
trade.Side = {'Buy'};
trade.Shares = 9300;
trade.Size = 0.06;
trade.Price = 29.68;
trade.ADV = 860000;
trade.Volatility = 0.27;
trade.POV = 0.17;
```

These examples do not represent real market data.

Data Types: `struct` | `table`

Output Arguments

`tr` — Timing risk

vector

Timing risk, returned as a vector. The vector values correspond to the timing risk in basis points for each stock in `trade`.

More About

Timing Risk

Timing risk (TR) estimates the uncertainty surrounding the estimated transaction cost.

Price volatility and liquidity risk creates uncertainty. Price volatility causes the price to be either higher or lower than expected due to factors independent of the order. Liquidity risk causes the market impact cost to be either higher or lower than estimated due to market volumes. TR is dependent upon volumes, intraday trading patterns, and market impact resulting from other market participants. The TR model is

$$\mathrm{TR} = \sigma \cdot \sqrt{\frac{1}{3} \cdot \frac{1}{250} \cdot \frac{Shares}{ADV} \cdot \left(\frac{1 - POV}{POV}\right)} \cdot 10^4 .$$

σ is price volatility. 250 is the number of trading days in the year. *Shares* are the number of shares to trade. *ADV* is the average daily volume of the stock. *POV* is the percentage of market volume, or participation fraction, of the order.

Tips

- For details about the formula and calculations, contact the Kissell Research Group.

Version History

Introduced in R2016a

References

[1] Kissell, Robert. “A Practical Framework for Transaction Cost Analysis.” *Journal of Trading*. Vol. 3, Number 2, Summer 2008, pp. 29–37.

[2] Kissell, Robert. “Algorithmic Trading Strategies.” Ph.D. Thesis. Fordham University, May 2006.

[3] Kissell, Robert. “TCA in the Investment Process: An Overview.” *Journal of Index Investing*. Vol. 2, Number 1, Summer 2011, pp. 60–64.

[4] Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*. Cambridge, MA: Elsevier/Academic Press, 2013.

[5] Glantz, Morton, and Robert Kissell. *Multi-Asset Risk Modeling*. Cambridge, MA: Elsevier/Academic Press, 2013.

[6] Kissell, Robert, and Morton Glantz. *Optimal Trading Strategies*. New York, NY: AMACOM, Inc., 2003.

See Also

`krg` | `iStar` | `marketImpact` | `priceAppreciation` | `liquidityFactor`

Topics

“Analyze Trading Execution Results” on page 6-2
“Estimate Portfolio Liquidation Costs” on page 6-20
“Conduct Sensitivity Analysis to Estimate Trading Costs” on page 6-17
“Optimize Percentage of Volume Trading Strategy” on page 6-23
“Optimize Trade Time Trading Strategy” on page 6-26
“Optimize Trade Schedule Trading Strategy” on page 6-29

moneynetws

Create Money.Net web socket interface connection

Description

The `moneynetws` function creates a `moneynetws` object. The `moneynetws` object represents a Money.Net web socket interface connection.

After you create a `moneynetws` object, you can use the object functions to retrieve current, intraday, historical, real-time, and news data. You retrieve data based on your credentials, which consist of a user name and password. For credentials, contact Money.Net.

Creation

Syntax

```
c = moneynetws(username,password)
```

Description

`c = moneynetws(username,password)` creates a Money.Net web socket interface connection, sets the Username property, and uses a password.

Input Arguments

`password` — Password
character vector | string scalar

Password required to access Money.Net data, specified as a character vector or string scalar. To request your Money.Net password, contact Money.Net.

Data Types: `char` | `string`

Properties

`Username` — User name
string scalar

User name required to access Money.Net data, specified as a string scalar. The user name is an email address. To request your Money.Net user name, contact Money.Net.

You can specify the user name as a string scalar or character vector in the `moneynetws` function.

Example: `"user@company.com"`

Data Types: `string`

Object Functions

Money.Net Connection

close	Close Money.Net connection
isconnection	Determine if Money.Net web socket interface connection is valid

Money.Net Data Retrieval

getdata	Retrieve Money.Net current data
getsubscriptions	Retrieve Money.Net subscribed symbols and event handler functions
news	Search and stream Money.Net latest news stories
optionchain	Retrieve Money.Net option symbols
realtime	Retrieve Money.Net real-time data
stop	Unsubscribe Money.Net real-time data updates
timeseries	Retrieve Money.Net intraday and historical data

Examples

Connect to Money.Net

Create a Money.Net web socket interface connection, and then retrieve current data for a symbol.

Connect to Money.Net using the Money.Net web socket interface and a user name and password. `c` is the Money.Net web socket interface connection object.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd)
```

```
c =

  moneynetws with properties:

    Username: "user@company.com"
```

Verify the Money.Net web socket interface connection `c` using the `isconnection` function. This function returns `1`, indicating a successful connection.

```
v = isconnection(c)
```

```
v =

  logical

   1
```

Retrieve Money.Net current data `d` for the symbol IBM by using the Money.Net web socket interface connection `c`. Specify the Money.Net data fields `f` for the highest and lowest prices of the current trading day.

```
symbol = "IBM";
f = ["high" "low"];
d = getdata(c,symbol,f);
```

Close the Money.Net web socket interface connection.

```
close(c)
```

Version History

Introduced in R2021b

See Also

Topics

“Retrieve Current and Historical Money.Net Data Using Money.Net Web Socket Interface” on page 7-2
“Retrieve Real-Time Money.Net Data Using Money.Net Web Socket Interface” on page 7-4
“Retrieve Money.Net News Stories Using Money.Net Web Socket Interface” on page 7-6

External Websites

Money.Net Help

close

Close Money.Net connection

Syntax

```
close(c)
```

Description

`close(c)` closes the Money.Net web socket interface connection `c`.

Examples

Close Money.Net Connection

Create a Money.Net web socket interface connection, retrieve current data for a symbol, and then close the connection.

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve Money.Net current data `d` for the symbol IBM by using the Money.Net web socket interface connection `c`. Specify the Money.Net data fields `f` for the highest and lowest prices of the current trading day.

```
symbol = "IBM";
f = ["high" "low"];
d = getdata(c,symbol,f);
```

Close the Money.Net web socket interface connection.

```
close(c)
```

Input Arguments

c — Money.Net web socket interface connection

`moneynetws` object

Money.Net web socket interface connection, specified as a `moneynetws` object created using the `moneynetws` function.

Version History

Introduced in R2021b

See Also

`moneynetws` | `isconnection` | `getdata` | `timeseries` | `realtime` | `news`

Topics

“Retrieve Current and Historical Money.Net Data Using Money.Net Web Socket Interface” on page 7-2
“Retrieve Real-Time Money.Net Data Using Money.Net Web Socket Interface” on page 7-4
“Retrieve Money.Net News Stories Using Money.Net Web Socket Interface” on page 7-6

getdata

Retrieve Money.Net current data

Syntax

```
d = getdata(c,symbols)
d = getdata(c,symbols,f)
```

Description

`d = getdata(c,symbols)` returns Money.Net data `d` using the Money.Net web socket interface connection `c` for the symbols.

`d = getdata(c,symbols,f)` returns Money.Net data for the specified Money.Net fields `f`.

Examples

Retrieve Money.Net Current Data

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve Money.Net current data `d` for the symbol IBM using the Money.Net web socket interface connection `c`. The table `d` contains the variables for all Money.Net fields.

```
symbol = "IBM";
d = getdata(c,symbol);
```

Close the Money.Net web socket interface connection.

```
close(c)
```

Retrieve Current Data for One Symbol

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve Money.Net current data `d` for the symbol IBM using the Money.Net web socket interface connection `c`. Specify the Money.Net data fields `f` for the highest and lowest prices of the current trading day. `d` is a table that contains the variables for high and low prices.

```
symbol = "IBM";
f = ["high" "low"];
d = getdata(c,symbol,f);
```

Close the Money.Net web socket interface connection.

```
close(c)
```

Retrieve Current Data for Multiple Symbols

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve Money.Net current data `d` for the `symbols` list that contains IBM and Google using the Money.Net web socket interface connection `c`. Specify the Money.Net data fields `f` for the highest and lowest prices of the current trading day.

```
symbols = ["IBM" "GOOG"];
f = ["high" "low"];
d = getdata(c,symbols,f);
```

`d` is a table that contains the variables for high and low prices. The rows contains the Money.Net data values for each symbol in the symbol list.

Close the Money.Net web socket interface connection.

```
close(c)
```

Input Arguments

`c` — Money.Net web socket interface connection

`moneynetws` object

Money.Net web socket interface connection, specified as a `moneynetws` object created using the `moneynetws` function.

`symbols` — Money.Net symbol list

character vector | cell array of character vectors | string scalar | string array

Money.Net symbol list, specified as a character vector, cell array of character vectors, string scalar, or a string array. To specify one symbol, use a character vector or string scalar. To specify multiple symbols, use a cell array of character vectors or a string array.

Example: `"IBM"`

Example: `["IBM" "GOOG"]`

Data Types: `char` | `cell` | `string`

`f` — Money.Net data field list

character vector | cell array of character vectors | string scalar | string array

Money.Net data field list, specified as a character vector, cell array of character vectors, string scalar, or a string array. To specify one field, use a character vector or string scalar. To specify multiple fields, use a cell array of character vectors or a string array.

Specify the field by using the field definition. For example, to specify the highest price for the equity during the current trading day, use the field definition `"high"`. The software ignores the case of the definition.

Example: `"high"`

Example: `["high" "low"]`

Data Types: `char` | `cell` | `string`

Output Arguments

d — Money.Net data

table

Money.Net data, returned as a table. Each row corresponds to the `symbols` list. Each variable corresponds to the field list `f`. The name of the rows in the table are the names of symbols.

Version History

Introduced in R2021b

See Also

`moneynetws` | `realtime` | `timeseries` | `news` | `close`

Topics

“Retrieve Current and Historical Money.Net Data Using Money.Net Web Socket Interface” on page 7-2

getsubscriptions

Retrieve Money.Net subscribed symbols and event handler functions

Syntax

```
subs = getsubscriptions(c)
```

Description

`subs = getsubscriptions(c)` returns the subscription list `subs` that contains open subscriptions for the Money.Net web socket interface connection `c`.

Examples

Retrieve Subscribed Symbols and Event Handlers

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Subscribe to the symbols IBM and Google for real-time data updates using the Money.Net connection `c`.

```
symbols = ["IBM" "GOOG"];
realtime(c,symbols)
```

The default event handler function processes real-time data updates.

Retrieve the subscribed symbols and the corresponding event handler function for each symbol using the Money.Net connection `c`.

```
subs = getsubscriptions(c)
```

```
subs =

  2×2 table

    Symbols           EventHandlers
    _______    ____________________________

    "GOOG"     {@mnWSRealTimeEventHandler}
    "IBM"      {@mnWSRealTimeEventHandler}
```

`subs` returns a table with a row for each symbol and the corresponding event handler function.

Unsubscribe from all symbols using the Money.Net connection `c`.

```
stop(c)
```

Close the Money.Net web socket interface connection.

```
close(c)
```

Input Arguments

c — Money.Net web socket interface connection

`moneynetws` object

Money.Net web socket interface connection, specified as a `moneynetws` object created using the `moneynetws` function.

Output Arguments

subs — Subscription list

table

Subscription list, returned as a table. The list contains all currently subscribed symbols and the corresponding event handler function that is processing real-time updates for each symbol. Each row in the table represents one unique subscription.

If there are no subscribed symbols, `subs` is an empty table.

Version History

Introduced in R2021b

See Also

`moneynetws` | `realtime` | `stop` | `close`

Topics

“Retrieve Real-Time Money.Net Data Using Money.Net Web Socket Interface” on page 7-4

isconnection

Determine if Money.Net web socket interface connection is valid

Syntax

```
v = isconnection(c)
```

Description

`v = isconnection(c)` returns logical `1 (true)` if `c` is a valid Money.Net web socket interface connection. Otherwise, `isconnection` returns logical `0 (false)`.

Examples

Validate Money.Net Web Socket Interface Connection

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Validate the Money.Net web socket interface connection `c`.

```
v = isconnection(c)
```

```
v =

  logical

   1
```

`isconnection` returns `1`, indicating a successful connection.

Close the Money.Net web socket interface connection.

```
close(c)
```

Validate that the Money.Net web socket interface connection `c` is closed.

```
v = isconnection(c)
```

```
v =

  logical

   0
```

`isconnection` returns `0`, indicating a closed connection.

Input Arguments

c — Money.Net web socket interface connection

`moneynetws` object

Money.Net web socket interface connection, specified as a `moneynetws` object created using the `moneynetws` function.

Output Arguments

v — Valid Money.Net web socket interface connection

`true` | `false`

Valid Money.Net web socket interface connection, returned as a logical `1 (true)` that specifies a successful connection, or logical `0 (false)` that specifies a closed or invalid connection.

Version History

Introduced in R2021b

See Also

`moneynetws` | `close`

Topics

“Retrieve Current and Historical Money.Net Data Using Money.Net Web Socket Interface” on page 7-2
“Retrieve Real-Time Money.Net Data Using Money.Net Web Socket Interface” on page 7-4
“Retrieve Money.Net News Stories Using Money.Net Web Socket Interface” on page 7-6

news

Search and stream Money.Net latest news stories

Syntax

```
n = news(c)
n = news(c,Name=Value)

news(c,Subscription="on",EventHandler=eventhandler)
news(c,Subscription="off")
```

Description

Retrieve Money.Net News Stories

`n = news(c)` returns Money.Net news stories `n` using the Money.Net web socket interface connection `c`.

`n = news(c,Name=Value)` specifies options using one or more name-value arguments. For example, `Number=25` returns 25 news stories.

Subscribe to Real-Time Updates for Money.Net News Stories

`news(c,Subscription="on",EventHandler=eventhandler)` subscribes to real-time updates for news stories. This syntax executes the specified event handler function when a new news story becomes available.

`news(c,Subscription="off")` stops the real-time updates for news stories.

Examples

Retrieve News Stories

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve news data `n` for 50 news stories using the Money.Net web socket interface connection `c`.

```
n = news(c);
```

`n` returns as a table with 50 rows.

Close the Money.Net web socket interface connection.

```
close(c)
```

Retrieve a Specific Number of Stories

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve news data `n` for 10 news stories using the Money.Net web socket interface connection `c`.

```
n = news(c,Number=10);
```

`n` returns as a table with 10 rows.

Close the Money.Net web socket interface connection.

```
close(c)
```

Filter News Story Retrieval by Specific Criteria

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve news stories in the general finance category. Specify that the news stories mention the term `"stock"` and contain the symbol for IBM.

```
category = "General Finance";
term = "stock";
symbol = "IBM";
n = news(c,Category=category,SearchTerm=term,Symbol=symbol);
```

`n` is a table with 50 news stories.

Close the Money.Net web socket interface connection.

```
close(c)
```

Stream News Data

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Turn on the subscription to the Money.Net real-time news data stream using the default event handler function `mnNewsStreamEventHandler`. The function `mnNewsStreamEventHandler` processes news data events by populating the workspace variable `mnNewsStreamLatest` with the

latest news stories. News stories populate in the `mnNewsStreamLatest` variable until it contains 10 rows. Then, the latest news stories overwrite the older ones in `mnNewsStreamLatest`. To access the code for this function, enter `edit mnNewsStreamEventHandler.m`.

```
eventhandler = "mnNewsStreamEventHandler";
news(c,Subscription="on",EventHandler=eventhandler)
```

The workspace variable `mnNewsStreamLatest` appears in the MATLAB Workspace. To see the latest 10 news stories, explore `mnNewsStreamLatest` in the Variables editor.

Turn off the real-time news data stream.

```
news(c,Subscription="off")
```

Real-time updates stop in the workspace variable `mnNewsStreamLatest`.

Close the Money.Net web socket interface connection.

```
close(c)
```

Stream News Data Using Custom Event Handler

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Turn on the subscription to the Money.Net real-time news data stream using the custom event handler function `myfnc`. Here, define `myfnc` to display Money.Net news data to the Command Window. You can write a custom event handler function to process streaming news stories differently. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
myfnc = @(x)disp(x);
news(c,Subscription="on",EventHandler=myfnc)
```

Money.Net news stories stream to the Command Window.

Turn off the real-time news data stream.

```
news(c,Subscription="off")
```

Real-time updates stop in the Command Window.

Close the Money.Net web socket interface connection.

```
close(c)
```

Input Arguments

c — Money.Net web socket interface connection

`moneynetws` object

Money.Net web socket interface connection, specified as a `moneynetws` object created using the `moneynetws` function.

`eventhandler` — Event handler

character vector | string scalar | function handle

Event handler, specified as a character vector, string scalar, or a function handle that specifies the name of the event handler function. Write a custom event handler function to process Money.Net events related to news stories. For details about custom event handler functions, see “Writing and Running Custom Event Handler Functions” on page 1-26.

Data Types: `char` | `function_handle` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `n = news(c,Number=10);` returns 10 Money.Net news stories.

Note The name-value arguments in the searching and streaming groups are independent. If you combine these name-value arguments, you receive this error: `Invalid combination of Name-Value pairs. Type HELP MONEYNET/NEWS to see the valid syntax`.

Searching News Stories Options

`Number` — Number of news stories

`50` (default) | numeric scalar

Number of news stories, specified as a numeric scalar. The maximum number of news stories that the Money.Net web socket interface can return is 500.

The number of news stories returned can be fewer than the specified number because Money.Net provides only available news stories. When you specify this option by itself, `news` does not filter the story content.

Example: `n = news(c,Number=10);`

Data Types: `double`

`SearchTerm` — Search term

character vector | string scalar

Search term, specified as a character vector or string scalar. `news` returns available news stories that contain the search term in the title or body of the news story.

Example: `n = news(c,SearchTerm="Windows 10");`

Data Types: `char` | `string`

`Symbol` — Symbol

character vector | cell array of character vectors | string scalar | string array

Symbol, specified as a character vector, cell array of character vectors, string scalar, or a string array. To specify one symbol, use a character vector or string scalar. To specify multiple symbols, use a cell

array of character vectors or a string array. `news` returns news stories related to the specified symbols.

Example: `n = news(c,Symbol=["IBM" "YHOO"]);`

Data Types: `char` | `cell` | `string`

`Category` — News category
character vector | string scalar

News category, specified as a character vector or a string scalar. `news` returns stories only in the news category specified.

Example: `n = news(c,Category="General Finance");`

Data Types: `char` | `string`

Streaming News Stories Options

`Subscription` — Money.Net real-time news subscription
`"on"` | `"off"`

Money.Net real-time news subscription, specified as the values `"on"` or `"off"`. To turn on the Money.Net real-time news subscription, specify the value `"on"`. To turn off the subscription, specify the value `"off"`.

By default, the sample event handler function `mnNewsStreamEventHandler` processes the retrieval of news stories during real-time news subscription. To access the code for this function, enter `edit mnNewsStreamEventHandler.m`. The `mnNewsStreamEventHandler` function creates the workspace variable `mnNewsStreamLatest`. Then, `mnNewsStreamEventHandler` populates the table `mnNewsStreamLatest` with the latest 10 news stories from Money.Net. The `mnNewsStreamEventHandler` function updates the list to display the latest news stories.

To specify a custom event handler function, use the name-value argument `EventHandler`.

Example: `news(c,Subscription="on")`

Example: `news(c,Subscription="on",EventHandler=myFcn)`

`EventHandler` — Custom event handler function
character vector | string scalar | function handle

Custom event handler function, specified as a character vector, string scalar, or function handle. To process the latest news stories, you can write your own custom event handler function. This function must have an input argument specified as a table. Each new news story from Money.Net is a single row in a table. For details about working with custom event handler functions, see “Writing and Running Custom Event Handler Functions” on page 1-26.

Specify this name-value argument only with the name-value argument `Subscription` and value `"on"`.

Example: `news(c,Subscription="on",EventHandler=myFcn)`

Data Types: `char` | `function_handle` | `string`

Output Arguments

n — News stories
table

News stories, returned as a table with these variables. Each row in the table represents one news story.

News Story Variable	Data Type	Variable Description
`id`	string array	News story identifier
`publisher`	table	Publisher name and identifier
`headline`	string array	News story headline
`content`	string array	Portion of news story content
`url`	string array	URL for the website that contains the news story
`tickers`	cell array	List of tickers associated with the news story
`posted`	`datetime` array	Date and time the news story was posted
`updated`	`datetime` array	Date and time the news story was last updated
`category`	string array	Category for the news story

Version History

Introduced in R2021b

See Also

`moneynetws` | `getdata` | `timeseries` | `realtime` | `close`

Topics

“Retrieve Money.Net News Stories Using Money.Net Web Socket Interface” on page 7-6
“Writing and Running Custom Event Handler Functions” on page 1-26

optionchain

Retrieve Money.Net option symbols

Syntax

```
o = optionchain(c,s)
o = optionchain(c,s,ExpiryType=type)
```

Description

`o = optionchain(c,s)` returns the option symbols using the Money.Net web socket interface connection `c` and symbol `s`.

`o = optionchain(c,s,ExpiryType=type)` returns the option symbols for the specified expiry type.

Examples

Retrieve Option Symbols for Specified Symbol

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve option symbols `o` for the symbol IBM.

```
s = "IBM";
o = optionchain(c,s);
```

`o` is a table. Each row of the table is an option symbol.

Retrieve current data for the first option symbol `o.symbol(1)`. Specify field `f` for describing the option symbol.

```
symbol = o.symbol(1);
f = ["Description"];
d = getdata(c,symbol,f);
```

`d` is a table with one row of data. The name of the row is the option symbol name. The variable contains the description for the option symbol.

To retrieve intraday data, use `timeseries`.

Close the Money.Net web socket interface connection.

```
close(c)
```

Input Arguments

c — Money.Net web socket interface connection

`moneynetws` object

Money.Net web socket interface connection, specified as a `moneynetws` object created using the `moneynetws` function.

s — Money.Net symbol

character vector | cell array of character vector | string scalar

Money.Net symbol, specified as a character vector, cell array of a character vector, or string scalar to denote one symbol.

Example: `"IBM"`

Data Types: `char` | `cell` | `string`

type — Expiry type

`"weekly"` | `"monthly"`

Expiry type of options, specified as the values `"weekly"` for weekly expiry or `"monthly"` for monthly expiry. You can specify these values as a string scalar or character vector.

Data Types: `char` | `string`

Output Arguments

o — Option symbols

table

Option symbols, returned as a table with these variables:

- `symbol` — Option symbols
- `exercise` — Exercise type (for example, `AMERICAN`)
- `indicator` — Option type (`CALL` or `PUT`)
- `strikePrice` — Strike price of the option
- `expiry` — Option expiration date

The total number of option symbols depends on the symbol `s`.

Version History

Introduced in R2021b

See Also

`moneynetws` | `getdata` | `realtime` | `timeseries` | `close`

Topics

“Retrieve Current and Historical Money.Net Data Using Money.Net Web Socket Interface” on page 7-2

realtime

Retrieve Money.Net real-time data

Syntax

```
realtime(c,symbols)
realtime(c,symbols,eventhandler)
```

Description

`realtime(c,symbols)` subscribes to real-time data updates using the Money.Net web socket interface connection `c` for the specified symbols. The default event handler function `mnWSRealTimeEventHandler` processes and retrieves real-time data updates for each specified symbol.

`realtime(c,symbols,eventhandler)` processes real-time data updates using a custom event handler function `eventhandler`.

Examples

Retrieve Money.Net Real-Time Data for One Symbol

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve Money.Net real-time data updates for the IBM symbol.

```
symbol = "IBM";
realtime(c,symbol)
```

The default event handler `mnWSRealTimeEventHandler` processes all real-time data updates. To access the code for the default event handler, enter `edit mnWSRealTimeEventHandler.m`.

`mnWSRealTimeEventHandler` creates the workspace variable `IBMRealTime`. The `mnWSRealTimeEventHandler` function populates the table `IBMRealTime` with real-time data updates. To see the real-time data, open `IBMRealTime` in the Variables editor.

Stop the symbol subscription.

```
stop(c)
```

`mnWSRealTimeEventHandler` stops processing all real-time data updates. The last real-time data update remains in `IBMRealTime`.

Close the Money.Net web socket interface connection.

```
close(c)
```

Retrieve Money.Net Real-Time Data for Multiple Symbols

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve Money.Net real-time data updates for the symbols IBM and Google.

```
symbols = ["IBM" "GOOG"];
realtime(c,symbols)
```

The default event handler `mnWSRealTimeEventHandler` processes all real-time data updates. To access the code for the default event handler, enter `edit mnWSRealTimeEventHandler.m`.

The `mnWSRealTimeEventHandler` function creates the workspace variables `IBMRealTime` and `GOOGRealTime`. The `mnWSRealTimeEventHandler` function populates the tables `IBMRealTime` and `GOOGRealTime` with real-time data updates. To see the real-time data, open either variable in the Variables editor.

Stop all symbol subscriptions.

```
stop(c)
```

`mnWSRealTimeEventHandler` stops processing all real-time data updates. The last real-time data update remains in each workspace variable.

Close the Money.Net web socket interface connection.

```
close(c)
```

Retrieve Money.Net Real-Time Data Using Custom Event Handler

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Define a custom event handler function `myfcn`. The `myfcn` function displays real-time Money.Net data to the Command Window. You can write a custom function that processes real-time data updates differently. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
myfcn = @(x)disp(x);
```

Retrieve Money.Net real-time data updates for the IBM symbol using `myfcn`.

```
symbol = "IBM";
realtime(c,symbol,myfcn)
```

`myfcn` displays real-time data updates for IBM in the Command Window.

Stop the symbol subscription.

```
stop(c)
```

`myfcn` stops displaying real-time data updates in the Command Window.

Close the Money.Net web socket interface connection.

```
close(c)
```

Input Arguments

`c` — Money.Net web socket interface connection

`moneynetws` object

Money.Net web socket interface connection, specified as a `moneynetws` object created using the `moneynetws` function.

`symbols` — Money.Net symbol list

character vector | cell array of character vectors | string scalar | string array

Money.Net symbol list, specified as a character vector, cell array of character vectors, string scalar, or a string array. To specify one symbol, use a character vector or string scalar. To specify multiple symbols, use a cell array of character vectors or a string array.

Example: `"IBM"`

Example: `["IBM" "GOOG"]`

Data Types: `char` | `cell` | `string`

`eventhandler` — Event handler

`"mnWSRealTimeEventHandler"` (default) | character vector | string scalar | function handle

Event handler, specified as a character vector, string scalar, or a function handle that specifies the name of the event handler function. Write a custom event handler function to process any type of real-time Money.Net events. This function must have at least one input argument that is a table. The table format must be similar to the format of the output argument in `getdata`. The event handler function returns all available fields when it executes for the first time. The event handler function executes every time Money.Net provides a real-time update. For details about custom event handler functions, see “Writing and Running Custom Event Handler Functions” on page 1-26.

For example, to display real-time data updates in the Command Window, enter this code to define a custom event handler function:

```
symbol = "IBM";
myfcn = @(x)disp(x);

realtime(c,symbol,myfcn)
```

If you do not specify a custom event handler function, the default event handler `mnWSRealTimeEventHandler` runs. To access the code for the default event handler, enter `edit mnWSRealTimeEventHandler.m`.

The `mnWSRealTimeEventHandler` function creates a workspace variable. The workspace variable name is a concatenation of the symbol name and the word `RealTime`. For example, `mnWSRealTimeEventHandler` populates real-time data for the symbol IBM into `IBMRealTime`. This workspace variable is a table with variables for each field. The values in the table change when Money.Net provides a real-time data update. Empty fields from Money.Net populate as `NaN`, `NaT`, and so on, depending on the data type.

First, `mnWSRealTimeEventHandler` runs using a table of current data. Then, `mnWSRealTimeEventHandler` runs each time an update occurs.

Data Types: `char` | `function_handle` | `string`

Version History

Introduced in R2021b

See Also

`moneynetws` | `getsubscriptions` | `stop` | `getdata` | `timeseries` | `news` | `close`

Topics

“Retrieve Real-Time Money.Net Data Using Money.Net Web Socket Interface” on page 7-4
“Writing and Running Custom Event Handler Functions” on page 1-26

stop

Unsubscribe Money.Net real-time data updates

Syntax

```
stop(c)
stop(c,symbols)
```

Description

`stop(c)` unsubscribes real-time data updates associated with the Money.Net web socket interface connection `c`.

`stop(c,symbols)` unsubscribes real-time data updates for the specified symbols.

Examples

Unsubscribe All Real-Time Data Updates

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Subscribe to the symbols IBM and Google for real-time data updates using the Money.Net connection `c`.

```
symbols = ["IBM" "GOOG"];
realtime(c,symbols)
```

The default event handler function processes real-time data updates.

Unsubscribe from all symbol subscriptions.

```
stop(c)
```

The default event handler function stops processing all real-time data updates. For details about the event handler function, see `realtime`.

Close the Money.Net web socket interface connection.

```
close(c)
```

Unsubscribe Real-Time Data Updates for One Symbol

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Subscribe to the symbols IBM and Google for real-time data updates using the Money.Net connection `c`.

```
symbols = ["IBM" "GOOG"];
realtime(c,symbols)
```

The default event handler function processes real-time data updates.

Unsubscribe from real-time data updates for IBM only.

```
symbol = "IBM";
stop(c,symbol)
```

The default event handler function stops processing real-time data updates for IBM. The real-time data updates continue for Google only. For details about the event handler function, see `realtime`.

Close the Money.Net web socket interface connection.

```
close(c)
```

Input Arguments

`c` — Money.Net web socket interface connection

`moneynetws` object

Money.Net web socket interface connection, specified as a `moneynetws` object created using the `moneynetws` function.

`symbols` — Money.Net symbol list

character vector | cell array of character vectors | string scalar | string array

Money.Net symbol list, specified as a character vector, cell array of character vectors, string scalar, or a string array. To specify one symbol, use a character vector or string scalar. To specify multiple symbols, use a cell array of character vectors or a string array.

Example: `"IBM"`

Example: `["IBM" "GOOG"]`

Data Types: `char` | `cell` | `string`

Version History

Introduced in R2021b

See Also

`moneynetws` | `getsubscriptions` | `realtime` | `close`

Topics

“Retrieve Real-Time Money.Net Data Using Money.Net Web Socket Interface” on page 7-4

timeseries

Retrieve Money.Net intraday and historical data

Syntax

```
d = timeseries(c,s,date,interval)
d = timeseries(c,s,date,interval,f)
```

Description

`d = timeseries(c,s,date,interval)` returns Money.Net intraday and historical data using the Money.Net web socket interface connection `c` for all available fields. Specify the Money.Net symbol `s` and the current or historical date. To specify the amount of data to return, use the bar interval.

`d = timeseries(c,s,date,interval,f)` returns Money.Net intraday and historical data for the specified Money.Net fields `f`.

Examples

Retrieve Intraday Data in Minutes

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve intraday data for yesterday in 5-minute bars for the symbol Google using the Money.Net web socket interface connection `c`. Specify the date as 1 hour ago using `datetime`. To retrieve data in 5-minute bars, specify the interval as `"5M"`.

```
s = "GOOG";
date = [datetime("now")-hours(1) datetime("now")];
interval = "5M";
d = timeseries(c,s,date,interval);
```

`d` is a table that contains these variables:

- Date timestamp
- High price
- Close price
- Low price
- Open price
- Trading volume

Close the Money.Net web socket interface connection.

```
close(c)
```

Retrieve Daily Historical Data for Specified Fields

Create Money.Net web socket interface connection `c` using a user name and password.

```
username = "user@company.com";
pwd = "999999";

c = moneynetws(username,pwd);
```

Retrieve historical data in daily bars for the symbol IBM using the Money.Net web socket interface connection `c`. Specify the date range from June 1, 2015, through June 5, 2015, using `datetime`. To retrieve daily data, specify the interval as `"1D"`. Retrieve only the high and low price fields `f` from Money.Net.

```
s = "IBM";
date = [datetime("1-Jun-2015") datetime("5-Jun-2015")];
interval = "1D";
f = ["high" "low"];
d = timeseries(c,s,date,interval,f);
```

`d` is a table that contains these variables:

- Date timestamp
- High price
- Low price

Close the Money.Net web socket interface connection.

```
close(c)
```

Input Arguments

c — Money.Net web socket interface connection

`moneynetws` object

Money.Net web socket interface connection, specified as a `moneynetws` object created using the `moneynetws` function.

s — Money.Net symbol

character vector | cell array of character vector | string scalar

Money.Net symbol, specified as a character vector, cell array of a character vector, or string scalar to denote one symbol.

Example: `"IBM"`

Data Types: `char` | `cell` | `string`

date — Date

`datetime` array | character vector | cell array of character vectors | double | string scalar | string array

Date, specified as a `datetime` array, character vector, cell array of character vectors, double, string scalar, or string array. If `date` contains one date, this date is the start date. The software determines the end date to be the last second of the same day. If `date` contains two dates, the first date is the start date and the second date is the end date.

Example: `datetime("yesterday")`

Data Types: `datetime` | `char` | `cell` | `double` | `string`

`interval` — Interval
character vector | string scalar

Interval between bars, specified as a character vector or string scalar. Specify the interval as a number followed by one of these letters: `S`, `M`, or `D`. These letters indicate seconds, minutes, and days, respectively. For example, `30M` is 30-minute bars and `1D` is daily end-of-day data.

Data Types: `char` | `string`

`f` — Money.Net data field list
character vector | cell array of character vectors | string scalar | string array

Money.Net data field list, specified as a character vector, cell array of character vectors, string scalar, or a string array. To specify one field, use a character vector or string scalar. To specify multiple fields, use a cell array of character vectors or a string array.

Specify the field by using the single character or the field definition. For example, to specify the highest price for the equity during the current trading day, use a single character `"H"` or the corresponding field definition `"high"`. When using the field definition, the software ignores the case of the definition.

Example: `"high"`

Example: `["high" "low"]`

Data Types: `char` | `cell` | `string`

Output Arguments

`d` — Money.Net data
timetable

Money.Net data, returned as a timetable. Each row in the table represents data at different times. The first variable `bar` is the timestamp. The remaining variables contain one variable of data for each Money.Net field `f`.

To return data for all available historical fields, use this syntax:

```
d = timeseries(c,s,date,interval);
```

Money.Net returns data only for business days with trading activity.

Version History

Introduced in R2021b

See Also

`moneynetws` | `getdata` | `realtime` | `news` | `close`

Topics

“Retrieve Current and Historical Money.Net Data Using Money.Net Web Socket Interface” on page 7-2

moneynet

Create Money.Net connection

Description

The `moneynet` function creates a `moneynet` object. The `moneynet` object represents a Money.Net connection.

After you create a `moneynet` object, you can use the object functions to retrieve current, intraday, historical, real-time, and news data. You retrieve data based on your credentials, which consist of a user name and password. For credentials, contact Money.Net.

Creation

Syntax

```
c = moneynet(username,password)
c = moneynet(username,password,portnumber)
```

Description

`c = moneynet(username,password)` creates a Money.Net connection, sets the Username property, and uses a password.

`c = moneynet(username,password,portnumber)` also sets the Port property.

Input Arguments

`password` — Password
character vector | string scalar

Password required to access Money.Net data, specified as a character vector or string scalar. To request your Money.Net password, contact Money.Net.

Data Types: `char` | `string`

Properties

`Username` — User name
character vector | string scalar

User name required to access Money.Net data, specified as a character vector or string scalar. The user name is an email address. To request your Money.Net user name, contact Money.Net.

Example: `'user@company.com'`

Data Types: `char` | `string`

`Port` — Port number
`50010` (default) | numeric scalar

Port number of the Money.Net data server, specified as a numeric scalar.

Data Types: `double`

`Server` — Money.Net server name
character vector

This property is read-only.

Money.Net server name, specified as a character vector.

Example: `'NTY_JAMES_IRWIN_88 TCP'`

Data Types: `char`

Object Functions

Money.Net Connection

close	Close Money.Net connection
isconnection	Determine if Money.Net connection is valid

Money.Net Data Retrieval

getdata	Retrieve Money.Net current data
getsubscriptions	Retrieve Money.Net subscribed symbols and event handler functions
news	Search and stream Money.Net latest news stories
optionchain	Retrieve Money.Net option symbols
realtime	Retrieve Money.Net real-time data
stop	Unsubscribe Money.Net real-time data updates
timeseries	Retrieve Money.Net intraday and historical data

Examples

Connect to Money.Net

Create a Money.Net connection, and then retrieve current data for a symbol.

Connect to Money.Net using a user name and password. `c` is the Money.Net connection object.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd)
```

```
c =

  moneynet with properties:

    Username: 'user@company.com'
        Port: 50010
      Server: 'NTY_JAMES_IRWIN_88 TCP'
```

Verify the Money.Net connection `c` using the `isconnection` function. This function returns `1`, indicating a successful connection.

```
v = isconnection(c)
```

```
v =

  logical

   1
```

Retrieve Money.Net current data `d` for the symbol IBM by using the Money.Net connection `c`. Specify the Money.Net data fields `f` for ask and bid price.

```
symbol = 'IBM';
f = {'Ask','Bid'};

d = getdata(c,symbol,f);
```

Close the Money.Net connection.

```
close(c)
```

Connect to Money.Net Using Port Number

Create a Money.Net connection, and then retrieve news stories.

Connect to Money.Net using a user name, password, and port number. `c` is the Money.Net connection object.

```
username = 'user@company.com';
pwd = '999999';
portnumber = 50010;

c = moneynet(username,pwd,portnumber)
```

```
c =

  moneynet with properties:

    Username: 'user@company.com'
        Port: 50010
      Server: 'NTY_JAMES_IRWIN_88 TCP'
```

Verify the Money.Net connection `c` using the `isconnection` function. This function returns `1`, indicating a successful connection.

```
v = isconnection(c)
```

```
v =

  logical

   1
```

Retrieve news data `n` for 10 news stories by using the Money.Net connection `c`.

```
n = news(c,'Number',10);
```

Close the Money.Net connection.

```
close(c)
```

Version History

Introduced in R2016b

See Also

Topics

“Retrieve Current and Historical Money.Net Data” on page 8-2
“Retrieve Real-Time Money.Net Data” on page 8-5
“Retrieve Money.Net News Stories” on page 8-7
“Money.Net Error and Warning Messages” on page 8-10

External Websites

Money.Net API Documentation
Money.Net Help

close

Close Money.Net connection

Syntax

```
close(c)
```

Description

`close(c)` closes the Money.Net connection `c`.

Examples

Close Money.Net Connection

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Close the Money.Net connection.

```
close(c)
```

Input Arguments

c — Money.Net connection

connection object

Money.Net connection, specified as a connection object created using `moneynet`.

Version History

Introduced in R2016b

See Also

`moneynet` | `getdata` | `news` | `realtime` | `timeseries` | `isconnection`

Topics

“Retrieve Current and Historical Money.Net Data” on page 8-2
“Retrieve Real-Time Money.Net Data” on page 8-5
“Retrieve Money.Net News Stories” on page 8-7
“Money.Net Error and Warning Messages” on page 8-10

External Websites
Money.Net API Documentation

isconnection

Determine if Money.Net connection is valid

Syntax

```
v = isconnection(c)
```

Description

`v = isconnection(c)` returns logical `1 (true)` if `c` is a valid Money.Net connection. Otherwise, `isconnection` returns logical `0 (false)`.

Examples

Validate Money.Net Connection

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Validate the Money.Net connection `c`.

```
v = isconnection(c)
```

```
v =

  logical

   1
```

`isconnection` returns `1`, indicating a successful connection.

Close the Money.Net connection.

```
close(c)
```

Validate that the Money.Net connection `c` is closed.

```
v = isconnection(c)
```

```
v =

  logical

   0
```

`isconnection` returns `0`, indicating a closed connection.

Input Arguments

c — Money.Net connection

connection object

Money.Net connection, specified as a connection object created using `moneynet`.

Output Arguments

v — Valid Money.Net connection

`true` | `false`

Valid Money.Net connection, returned as a logical `1 (true)` that specifies a successful connection, or logical `0 (false)` that specifies a closed or invalid connection.

Version History

Introduced in R2016b

See Also

`moneynet` | `close`

Topics

“Retrieve Current and Historical Money.Net Data” on page 8-2
“Retrieve Real-Time Money.Net Data” on page 8-5
“Retrieve Money.Net News Stories” on page 8-7
“Money.Net Error and Warning Messages” on page 8-10

External Websites

Money.Net API Documentation

getdata

Retrieve Money.Net current data

Syntax

```
d = getdata(c,symbols,f)
```

Description

`d = getdata(c,symbols,f)` returns Money.Net data `d` using the Money.Net connection `c` for the symbols and the Money.Net fields `f`.

Examples

Retrieve Current Data for One Symbol

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve Money.Net current data `d` for the symbol IBM using the Money.Net connection `c`. Specify the Money.Net data fields `f` for ask and bid price.

```
symbol = 'IBM';
f = {'Ask','Bid'};

d = getdata(c,symbol,f);
```

Display Money.Net current data.

```
d
```

```
d =

    Symbol      Ask        Bid
    ______    _______    _______

    'IBM'     145.00     143.85
```

`d` is a table that contains the columns for symbol, ask price, and bid price. The row contains the Money.Net data values for each column.

Close the Money.Net connection.

```
close(c)
```

Retrieve Current Data for Multiple Symbols

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve Money.Net current data `d` for the `symbols` list that contains IBM, Google, and Yahoo! using the Money.Net connection `c`. Specify the Money.Net data fields `f` for ask and bid price.

```
symbols = {'IBM','GOOG','YHOO'};
f = {'Ask','Bid'};

d = getdata(c,symbols,f);
```

Display Money.Net current data.

```
d
```

```
d =

    Symbol      Ask        Bid
    ______    _______    _______

    'IBM'     145.00     143.85
    'GOOG'    700.50     700.05
    'YHOO'     37.50      37.41
```

`d` is a table that contains the columns for symbol, ask price, and bid price. The rows contains the Money.Net data values for each symbol in the symbol list.

Close the Money.Net connection.

```
close(c)
```

Input Arguments

`c` — Money.Net connection

connection object

Money.Net connection, specified as a connection object created using `moneynet`.

`symbols` — Money.Net symbol list

character vector | cell array of character vectors | string scalar | string array

Money.Net symbol list, specified as a character vector, cell array of character vectors, string scalar, or a string array. To specify one symbol, use a character vector or string scalar. To specify multiple symbols, use a cell array of character vectors or a string array.

Example: `'IBM'`

Example: `{'IBM','GOOG'}`

Data Types: `char` | `cell` | `string`

`f` — Money.Net data field list

character vector | cell array of character vectors | string scalar | string array

Money.Net data field list, specified as a character vector, cell array of character vectors, string scalar, or a string array. To specify one field, use a character vector or string scalar. To specify multiple fields, use a cell array of character vectors or a string array.

Specify the field by using the single character or the field definition. For example, to specify the highest price for the equity during the current trading day, use a single character `"H"` or the corresponding field definition `"High"`. When using the field definition, the software ignores the case of the definition. To view the list of valid Money.Net fields and field definitions, see the Money.Net API Documentation.

Example: `"High"`

Example: `["High" "Low"]`

Data Types: `char` | `cell` | `string`

Output Arguments

`d` — Money.Net data

table

Money.Net data, returned as a table. Each row corresponds to the `symbols` list. Each column corresponds to the field list `f`.

Version History

Introduced in R2016b

See Also

`moneynet` | `news` | `realtime` | `timeseries` | `close`

Topics

“Retrieve Current and Historical Money.Net Data” on page 8-2
“Money.Net Error and Warning Messages” on page 8-10

External Websites

Money.Net API Documentation

timeseries

Retrieve Money.Net intraday and historical data

Syntax

```
d = timeseries(c,s,date,interval)
d = timeseries(c,s,date,interval,f)
```

Description

`d = timeseries(c,s,date,interval)` returns Money.Net intraday and historical data using the Money.Net connection `c` for all available fields. Specify the Money.Net symbol `s` and the current or historical date. To specify the amount of data to return, use the bar interval.

`d = timeseries(c,s,date,interval,f)` returns Money.Net intraday and historical data for the specified Money.Net fields `f`.

Examples

Retrieve Intraday Data in Seconds

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve intraday data for the last 5 minutes in 30-second bars for the symbol IBM using the Money.Net connection `c`. Specify the date as a `datetime` array containing a date range with start and end dates. The start date starts 5 minutes after the current moment. The end date is the current moment. To specify the current moment, use `datetime('now')`. To specify 5 minutes earlier, subtract `minutes(5)` from the current moment. To retrieve data in 30-second bars, specify the interval as `'30S'`.

```
s = 'IBM';
date = [datetime('now')-minutes(5) datetime('now')];
interval = '30S';

d = timeseries(c,s,date,interval);
```

Display the first three rows of intraday data `d` for all valid Money.Net fields.

```
d(1:3,:)
```

```
ans =

          Date            High      Low       Open      Close     Volume
    __________________    ______    ______    ______    ______    _______
```

```
    05/09/16 13:30:30    147.52    147.48    147.48    147.51    2763.00
    05/09/16 13:31:00    147.53    147.50    147.50    147.52    7241.00
    05/09/16 13:31:30    147.54    147.51    147.51    147.53    5608.00
```

`d` is a table that contains these columns:

- Date timestamp
- High price
- Low price
- Open price
- Close price
- Trading volume

Close the Money.Net connection.

```
close(c)
```

Retrieve Intraday Data in Minutes

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve intraday data for yesterday in 30-minute bars for the symbol IBM using the Money.Net connection `c`. Specify the date as yesterday using `datetime`. To retrieve data in 30-minute bars, specify the interval as `'30M'`.

```
s = 'IBM';
date = datetime('yesterday');
interval = '30M';

d = timeseries(c,s,date,interval);
```

Display the first three rows of intraday data `d` for all valid Money.Net fields.

```
d(1:3,:)
```

```
ans =

          Date             High      Low       Open      Close     Volume
    ___________________    ______    ______    ______    ______    _______

    05/06/16 08:00:00    145.22    145.07    145.07    145.22    2455.00
    05/06/16 08:30:00    144.66    144.66    144.66    144.66     300.00
    05/06/16 09:00:00    145.00    144.90    144.90    145.00    4758.00
```

`d` is a table that contains these columns:

- Date timestamp
- High price

- Low price
- Open price
- Close price
- Trading volume

Close the Money.Net connection.

```
close(c)
```

Retrieve Daily Historical Data for Specified Fields

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve historical data in daily bars for the symbol IBM using the Money.Net connection `c`. Specify the date range from June 1, 2015, through June 5, 2015, using `datetime`. To retrieve daily data, specify the interval as `'1D'`. Retrieve only the high and low price fields `f` from Money.Net.

```
s = 'IBM';
date = [datetime('1-Jun-2015') datetime('5-Jun-2015')];
interval = '1D';
f = {'High','Low'};

d = timeseries(c,s,date,interval,f);
```

Display the first three rows of daily data `d`.

```
d(1:3,:)

ans =

          Date            High      Low
    ____________________    ______    ______

    06/01/15 00:00:00    171.04    169.03
    06/02/15 00:00:00    170.45    168.43
    06/03/15 00:00:00    171.56    169.63
```

`d` is a table that contains these columns:

- Date timestamp
- High price
- Low price

Close the Money.Net connection.

```
close(c)
```

Retrieve Weekly Historical Data for Specified Fields

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve historical data in weekly bars for the symbol IBM using the Money.Net connection `c`. Specify the date range from June 1, 2015 through June 30, 2015 using `datetime`. To retrieve weekly data, specify the interval as `'7D'`. Retrieve only the high and low price fields `f` from Money.Net.

```
s = 'IBM';
date = [datetime('1-Jun-2015') datetime('30-Jun-2015')];
interval = '7D';
f = {'High','Low'};

d = timeseries(c,s,date,interval,f);
```

Display the first three rows of weekly data `d`.

```
d(1:3,:)
```

```
ans =

          Date             High      Low
    ___________________    ______    ______

    06/01/15 00:00:00    171.56    167.20
    06/08/15 00:00:00    170.44    163.37
    06/15/15 00:00:00    168.72    164.25
```

`d` is a table that contains these columns:

- Date timestamp
- High price
- Low price

Close the Money.Net connection.

```
close(c)
```

Input Arguments

`c` — Money.Net connection

connection object

Money.Net connection, specified as a connection object created using `moneynet`.

`s` — Money.Net symbol

character vector | cell array of character vector | string scalar

Money.Net symbol, specified as a character vector, cell array of a character vector, or string scalar to denote one symbol.

Example: `"IBM"`

Data Types: `char` | `cell` | `string`

`date` — Date
`datetime` array | character vector | cell array of character vectors | double | string scalar | string array

Date, specified as a `datetime` array, character vector, cell array of character vectors, double, string scalar, or string array. If `date` contains one date, this date is the start date. The software determines the end date to be the last second of the same day. If `date` contains two dates, the first date is the start date and the second date is the end date.

Example: `datetime('yesterday')`

Data Types: `datetime` | `char` | `cell` | `double` | `string`

`interval` — Interval
character vector | string scalar

Interval between bars, specified as a character vector or string scalar. Specify the interval as a number followed by one of these letters: `S`, `M`, and `D`. These letters indicate seconds, minutes, and days, respectively. For example, `30S` is 30-second bars and `1D` is daily end-of-day data.

Data Types: `char` | `string`

`f` — Money.Net data field list
character vector | cell array of character vectors | string scalar | string array

Money.Net data field list, specified as a character vector, cell array of character vectors, string scalar, or a string array. To specify one field, use a character vector or string scalar. To specify multiple fields, use a cell array of character vectors or a string array.

Specify the field by using the single character or the field definition. For example, to specify the highest price for the equity during the current trading day, use a single character `"H"` or the corresponding field definition `"High"`. When using the field definition, the software ignores the case of the definition. To view the list of valid Money.Net fields and field definitions, see the Money.Net API Documentation.

Example: `"High"`

Example: `["High" "Low"]`

Data Types: `char` | `cell` | `string`

Output Arguments

`d` — Money.Net data
table

Money.Net data, returned as a table. Each row in the table represents data at different times. The first column `Date` is the timestamp. The remaining columns contain one column of data for each Money.Net field `f`.

To return data for all available historical fields, use this syntax:

```
d = timeseries(c,s,date,interval);
```

Money.Net returns data only for business days with trading activity.

Version History

Introduced in R2016b

See Also

`moneynet` | `getdata` | `news` | `realtime` | `close`

Topics

“Retrieve Current and Historical Money.Net Data” on page 8-2
“Money.Net Error and Warning Messages” on page 8-10

External Websites

Money.Net API Documentation

realtime

Retrieve Money.Net real-time data

Syntax

```
realtime(c,symbols)
realtime(c,symbols,eventhandler)
```

Description

`realtime(c,symbols)` subscribes to real-time data updates using the Money.Net connection `c` for the specified symbols. The default event handler function `mnRealTimeEventHandler` processes and retrieves real-time data updates for each specified symbol.

`realtime(c,symbols,eventhandler)` processes real-time data updates using a custom event handler function `eventhandler`.

Examples

Retrieve Money.Net Real-Time Data for One Symbol

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve Money.Net real-time data updates for the IBM symbol.

```
symbol = 'IBM';

realtime(c,symbol)
```

The default event handler `mnRealTimeEventHandler` processes all real-time data updates. To access the code for the default event handler, enter `edit mnRealTimeEventHandler.m`.

`mnRealTimeEventHandler` creates the workspace variable `IBMRealTime`. The `mnRealTimeEventHandler` function populates the table `IBMRealTime` with real-time data updates. To see the real-time data, open `IBMRealTime` in the Variables editor.

Stop the symbol subscription.

```
stop(c)
```

`mnRealTimeEventHandler` stops processing all real-time data updates. The last real-time data update remains in `IBMRealTime`.

Close the Money.Net connection.

```
close(c)
```

Retrieve Money.Net Real-Time Data for Multiple Symbols

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve Money.Net real-time data updates for the symbols IBM and Yahoo!.

```
symbols = {'IBM','YHOO'};

realtime(c,symbols)
```

The default event handler `mnRealTimeEventHandler` processes all real-time data updates. To access the code for the default event handler, enter `edit mnRealTimeEventHandler.m`.

The `mnRealTimeEventHandler` function creates the workspace variables `IBMRealTime` and `YHOORealTime`. The `mnRealTimeEventHandler` function populates the tables `IBMRealTime` and `YHOORealTime` with real-time data updates. To see the real-time data, open either variable in the Variables editor.

Stop all symbol subscriptions.

```
stop(c)
```

`mnRealTimeEventHandler` stops processing all real-time data updates. The last real-time data update remains in each workspace variable.

Close the Money.Net connection.

```
close(c)
```

Retrieve Money.Net Real-Time Data Using a Custom Event Handler

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Define a custom event handler function `myfcn`. The `myfcn` function displays real-time Money.Net data to the Command Window. You can write a custom function that processes real-time data updates differently. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
myfcn = @(x)disp(x);
```

Retrieve Money.Net real-time data updates for the IBM symbol using `myfcn`.

```
symbol = 'IBM';

realtime(c,symbol,myfcn)
```

Symbol	Description	Yesterday	YesterdayDateTime	Bid	Ask	ExchangeOfTheCurrentBidPrice	Ex
'IBM'	'INTERNATIONAL BUSINESS MACHS'	148.31	05/24/16 00:00:00	151.65	151.67	''	''

`myfcn` displays real-time data updates for IBM in the Command Window.

Stop the symbol subscription.

```
stop(c)
```

`myfcn` stops displaying real-time data updates in the Command Window.

Close the Money.Net connection.

```
close(c)
```

Input Arguments

`c` — Money.Net connection

connection object

Money.Net connection, specified as a connection object created using `moneynet`.

`symbols` — Money.Net symbol list

character vector | cell array of character vectors | string scalar | string array

Money.Net symbol list, specified as a character vector, cell array of character vectors, string scalar, or a string array. To specify one symbol, use a character vector or string scalar. To specify multiple symbols, use a cell array of character vectors or a string array.

Example: `'IBM'`

Example: `{'IBM','GOOG'}`

Data Types: `char` | `cell` | `string`

`eventhandler` — Event handler

`'mnRealTimeEventHandler'` (default) | character vector | string scalar | function handle

Event handler, specified as a character vector, string scalar, or a function handle that specifies the name of the event handler function. Write a custom event handler function to process any type of real-time Money.Net events. This function must have at least one input argument that is a table. The table format must be similar to the format of the output argument in `getdata`. The event handler function returns all available fields when it executes for the first time. The event handler function executes every time Money.Net provides a real-time update. For details about custom event handler functions, see “Writing and Running Custom Event Handler Functions” on page 1-26.

For example, to display real-time data updates in the Command Window, enter this code to define a custom event handler function:

```
symbol = 'IBM';
myfcn = @(x)disp(x);

realtime(c,symbol,myfcn)
```

If you do not specify a custom event handler function, the default event handler `mnRealTimeEventHandler` runs. To access the code for the default event handler, enter `edit mnRealTimeEventHandler.m`.

The `mnRealTimeEventHandler` function creates a workspace variable. The workspace variable name is a concatenation of the symbol name and the word `RealTime`. For example, `mnRealTimeEventHandler` populates real-time data for the symbol IBM into `IBMRealTime`. This workspace variable is a table with columns for each field. The values in the table change when Money.Net provides a real-time data update. Empty fields from Money.Net populate as `NaN`, `NaT`, and so on, depending on the data type.

First, `mnRealTimeEventHandler` runs using a table of current data. Then, `mnRealTimeEventHandler` runs each time an update occurs.

Data Types: `char` | `function_handle` | `string`

Version History

Introduced in R2016b

See Also

`moneynet` | `getdata` | `getsubscriptions` | `news` | `stop` | `close`

Topics

“Retrieve Real-Time Money.Net Data” on page 8-5
“Writing and Running Custom Event Handler Functions” on page 1-26
“Money.Net Error and Warning Messages” on page 8-10

External Websites

Money.Net API Documentation

stop

Unsubscribe Money.Net real-time data updates

Syntax

```
stop(c)
stop(c,symbols)
```

Description

`stop(c)` unsubscribes real-time data updates associated with the Money.Net connection `c`.

`stop(c,symbols)` unsubscribes real-time data updates for the specified symbols.

Examples

Unsubscribe All Real-Time Data Updates

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Subscribe to the symbols IBM and Yahoo! for real-time data updates using the Money.Net connection `c`.

```
symbols = ["IBM" "YHOO"];

realtime(c,symbols)
```

The default event handler function processes real-time data updates.

Unsubscribe from all symbol subscriptions.

```
stop(c)
```

The default event handler function stops processing all real-time data updates. For details about the event handler function, see `realtime`.

Close the Money.Net connection.

```
close(c)
```

Unsubscribe Real-Time Data Updates for One Symbol

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Subscribe to the symbols IBM and Yahoo! for real-time data updates using the Money.Net connection `c`.

```
symbols = ["IBM" "YHOO"];

realtime(c,symbols)
```

The default event handler function processes real-time data updates.

Unsubscribe from real-time data updates for IBM only.

```
symbol = 'IBM';

stop(c,symbol)
```

The default event handler function stops processing real-time data updates for IBM. The real-time data updates continue for Yahoo! only. For details about the event handler function, see `realtime`.

Close the Money.Net connection.

```
close(c)
```

Input Arguments

`c` — Money.Net connection

connection object

Money.Net connection, specified as a connection object created using `moneynet`.

`symbols` — Money.Net symbol list

character vector | cell array of character vectors | string scalar | string array

Money.Net symbol list, specified as a character vector, cell array of character vectors, string scalar, or a string array. To specify one symbol, use a character vector or string scalar. To specify multiple symbols, use a cell array of character vectors or a string array.

Example: `'IBM'`

Example: `{'IBM','GOOG'}`

Data Types: `char` | `cell` | `string`

Version History

Introduced in R2016b

See Also

`moneynet` | `getsubscriptions` | `realtime` | `close`

Topics

“Retrieve Real-Time Money.Net Data” on page 8-5

“Money.Net Error and Warning Messages” on page 8-10

External Websites
Money.Net API Documentation

getsubscriptions

Retrieve Money.Net subscribed symbols and event handler functions

Syntax

```
subs = getsubscriptions(c)
```

Description

`subs = getsubscriptions(c)` returns the subscription list `subs` that contains open subscriptions for the Money.Net connection `c`.

Examples

Retrieve Subscribed Symbols and Event Handlers

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Subscribe to the symbols IBM and Yahoo! for real-time data updates using the Money.Net connection `c`.

```
symbols = ["IBM" "YHOO"];

realtime(c,symbols)
```

The default event handler function processes real-time data updates.

Retrieve the subscribed symbols and the corresponding event handler function for each symbol using the Money.Net connection `c`.

```
subs = getsubscriptions(c)
```

```
subs =

    Symbols         EventHandlers
    _______    ____________________________

    'IBM'      @mnRealTimeEventHandler
    'YHOO'     @mnRealTimeEventHandler
```

`subs` returns a table with a row for each symbol and the corresponding event handler function.

Unsubscribe from all symbols using the Money.Net connection `c`.

```
stop(c)
```

Close the Money.Net connection.

```
close(c)
```

Input Arguments

c — Money.Net connection

connection object

Money.Net connection, specified as a connection object created using `moneynet`.

Output Arguments

subs — Subscription list

table

Subscription list, returned as a table. The list contains all currently subscribed symbols and the corresponding event handler function that is processing real-time updates for each symbol. Each row in the table represents one unique subscription.

If there are no subscribed symbols, `subs` is an empty table.

Version History

Introduced in R2016b

See Also

`moneynet` | `realtime` | `stop` | `close`

Topics

“Retrieve Real-Time Money.Net Data” on page 8-5
“Money.Net Error and Warning Messages” on page 8-10

External Websites

Money.Net API Documentation

optionchain

Retrieve Money.Net option symbols

Syntax

```
o = optionchain(c,s)
```

Description

`o = optionchain(c,s)` returns the option symbols using the Money.Net connection `c` and symbol `s`.

Examples

Retrieve Option Symbols for Specified Symbol

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve option symbols `o` for the symbol IBM.

```
s = 'IBM';

o = optionchain(c,s);
```

`o` is a cell array of character vectors. Each character vector is an option symbol.

Display the first three option symbols.

```
o(1:3)
```

```
ans =

  3×1 cell array

    'O:IBM\16Q13\130 .0'
    'O:IBM\16E27\148 .0'
    'O:IBM\16Q20\138 .0'
```

Retrieve current data for the first option symbol `o(1)` and display it. Specify fields `f` for describing the option symbol:

- Option symbol description
- Option symbol strike
- Option symbol expiration date

```
symbol = o(1);
f = {'Description','Strike','Expiration'};

d = getdata(c,symbol,f)
```

```
d =

          Symbol                 Description            Strike    Expiration
    ____________________    ____________________________    ______    __________

    'O:IBM\16F24\131 .0'    'IBM Call 06/24/2016 131.0'    131       06/24/16
```

`d` is a table with one row of data. The data contains the option symbol name in the first column and a column for each specified field `f`.

To retrieve intraday data, use `timeseries`.

Close the Money.Net connection.

```
close(c)
```

Input Arguments

c — Money.Net connection

connection object

Money.Net connection, specified as a connection object created using `moneynet`.

s — Money.Net symbol

character vector | cell array of character vector | string scalar

Money.Net symbol, specified as a character vector, cell array of a character vector, or string scalar to denote one symbol.

Example: `"IBM"`

Data Types: `char` | `cell` | `string`

Output Arguments

o — Option symbols

cell array of character vectors

Option symbols, returned as a cell array of character vectors. Each character vector specifies one option symbol. The total number of option symbols depends on the symbol `s`.

Version History

Introduced in R2016b

See Also

`moneynet` | `getdata` | `timeseries` | `close`

Topics

“Retrieve Current and Historical Money.Net Data” on page 8-2
“Money.Net Error and Warning Messages” on page 8-10

External Websites
Money.Net API Documentation

news

Search and stream Money.Net latest news stories

Syntax

```
n = news(c)
n = news(c,Name,Value)
news(c,Name,Value)
```

Description

`n = news(c)` returns Money.Net news stories `n` using the Money.Net connection `c`.

`n = news(c,Name,Value)` returns news stories with additional options specified by one or more `Name,Value` pair arguments.

`news(c,Name,Value)` streams news stories in real time using the streaming options.

Examples

Retrieve News Stories

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve news data `n` for 50 news stories using the Money.Net connection `c`.

```
n = news(c);
```

`n` returns as a table with 50 rows.

Display the news story title, identifier, and published time for the first news story in the table `n`.

```
n(1,1:3)
```

```
ans =

                          ArticleTitle                           ArticleID       PublishedTime
    _____________________________________________________________    _________    ________________

    'Stop talking about replacements. Give PC owners something new al…'    3.8917e+09    05/13/16 10:00:02
```

Close the Money.Net connection.

```
close(c)
```

Retrieve a Specific Number of Stories

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve news data `n` for 10 news stories using the Money.Net connection `c`.

```
n = news(c,'Number',10);
```

`n` returns as a table with 10 rows.

Display the news story title, identifier, and published time for the first news story in the table `n`.

```
n(1,1:3)
```

```
ans =
                              ArticleTitle                                 ArticleID       PublishedTime
    ___________________________________________________________________    _________    ________________

    'Stop talking about replacements. Give PC owners something new al…'    3.8917e+09    05/13/16 10:00:02
```

Close the Money.Net connection.

```
close(c)
```

Filter News Story Retrieval by Specific Criteria

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Retrieve news stories in the general finance category. Specify that the news stories mention the term `'Dropbox'` and contain the symbol for IBM.

```
category = 'General Finance';
term = 'Dropbox';
symbol = 'IBM';

n = news(c,'Category',category,'SearchTerm',term,'Symbol',symbol);
```

`n` is a table with one news story.

Display the news story title, identifier, and published time for the news story.

```
n(1,1:3)
```

```
ans =
                        ArticleTitle                          ArticleID       PublishedTime
    ____________________________________________________    _________    ________________

    'Hewlett Packard Enterprise (HPE) Teams Up with Dropbox'    4.0002e+09    06/08/16 11:11:05
```

Close the Money.Net connection.

```
close(c)
```

Stream News Data

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Turn on the subscription to the Money.Net real-time news data stream using the default event handler function `mnNewsStreamEventHandler`. The function `mnNewsStreamEventHandler` processes news data events by populating the workspace variable `mnNewsStreamLatest` with the latest news stories. News stories populate in the `mnNewsStreamLatest` variable until it contains 10 rows. Then, the latest news stories overwrite the older ones in `mnNewsStreamLatest`. To access the code for this function, enter `edit mnNewsStreamEventHandler.m`.

```
news(c,'Subscription','on')
```

The workspace variable `mnNewsStreamLatest` appears in the MATLAB Workspace.

Display the news story title, identifier, and published time for the first news story.

```
mnNewsStreamLatest(1,1:3)
```

```
ans =

                          ArticleTitle                                ArticleID       PublishedTime
    ______________________________________________________________    _________    ________________

    'Stop talking about replacements. Give PC owners something new al…'    3.8917e+09    05/13/16 10:00:02
```

To see the latest 10 news stories, explore `mnNewsStreamLatest` in the Variables editor.

Turn off the real-time news data stream.

```
news(c,'Subscription','off')
```

Real-time updates stop in the workspace variable `mnNewsStreamLatest`.

Close the Money.Net connection.

```
close(c)
```

Stream News Data Using Custom Event Handler

Create Money.Net connection `c` using a user name and password.

```
username = 'user@company.com';
pwd = '999999';

c = moneynet(username,pwd);
```

Turn on the subscription to the Money.Net real-time news data stream using the custom event handler function `myfnc`. Here, define `myfnc` to display Money.Net news data to the Command Window. You can write a custom event handler function to process streaming news stories differently. For details, see “Writing and Running Custom Event Handler Functions” on page 1-26.

```
myfnc = @(x)disp(x);

news(c,'Subscription','on','EventHandler',myfnc)
```

```
                          ArticleTitle                                ArticleID      PublishedTime      PublisherCode    Publisher
    ______________________________________________________________    _________    ________________    _____________    _________

    '@ETFcom: The Most Important ETF Of 2016 https://t.co/a5qCYK2o7c …'    3.9089e+09    05/17/16 14:39:10    'TWIT'           'Twitter'
```

Money.Net news stories stream to the Command Window.

Turn off the real-time news data stream.

```
news(c,'Subscription','off')
```

Real-time updates stop in the Command Window.

Close the Money.Net connection.

```
close(c)
```

Input Arguments

`c` — Money.Net connection

connection object

Money.Net connection, specified as a connection object created using `moneynet`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `n = news(c,'Number',10);`

Note The name-value pair arguments in the searching and streaming groups are independent. If you combine these name-value pair arguments, you receive this error: `Invalid combination of Name-Value pairs. Type HELP MONEYNET/NEWS to see the valid syntax`.

Searching News Stories Options

`Number` — Number of news stories

`50` (default) | numeric scalar

Number of news stories, specified as the comma-separated pair consisting of `'Number'` and a numeric scalar. The maximum number of news stories that the Money.Net API can return is 500.

The number of news stories returned can be fewer than the specified number because Money.Net provides only available news stories. When you specify this option by itself, `news` does not filter the story content.

Example: `n = news(c,'Number',10);`

Data Types: `double`

`SearchTerm` — Search term
character vector | string scalar

Search term, specified as the comma-separated pair consisting of `'SearchTerm'` and a character vector or a string scalar. `news` returns available news stories that contain the search term in the title or body of the news story.

Example: `n = news(c,'SearchTerm','Windows 10');`

Data Types: `char` | `string`

`Symbol` — Symbol
character vector | cell array of character vectors | string scalar | string array

Symbol, specified as the comma-separated pair consisting of `'Symbol'` and a character vector, cell array of character vectors, string scalar, or a string array. To specify one symbol, use a character vector or string scalar. To specify multiple symbols, use a cell array of character vectors or a string array. `news` returns news stories related to the specified symbols.

Example: `n = news(c,'Symbol',{'IBM,'YHOO'});`

Data Types: `char` | `cell` | `string`

`Category` — News category
character vector | string scalar

News category, specified as the comma-separated pair consisting of `'Category'` and a character vector or a string scalar. `news` returns stories only in the news category specified.

Example: `n = news(c,'Category','General Finance');`

Data Types: `char` | `string`

Streaming News Stories Options

`Subscription` — Money.Net real-time news subscription
`'on'` | `'off'`

Money.Net real-time news subscription, specified as the comma-separated pair consisting of `'Subscription'` and the values `'on'` or `'off'`. To turn on the Money.Net real-time news subscription, specify the value `'on'`. To turn off the subscription, specify the value `'off'`.

By default, the sample event handler function `mnNewsStreamEventHandler` processes the retrieval of news stories during real-time news subscription. To access the code for this function, enter `edit mnNewsStreamEventHandler.m`. The `mnNewsStreamEventHandler` function creates the workspace variable `mnNewsStreamLatest`. Then, `mnNewsStreamEventHandler` populates the table `mnNewsStreamLatest` with the latest 10 news stories from Money.Net. Then, the `mnNewsStreamEventHandler` function updates the list to display the latest news stories.

To specify a custom event handler function, use the name-value pair argument `'EventHandler'`.

Example: `news(c,'Subscription','on')`

Example: `news(c,'Subscription','on','EventHandler',myFcn)`

`EventHandler` — Custom event handler function
character vector | string scalar | function handle

Custom event handler function, specified as the comma-separated pair consisting of `'EventHandler'` and a character vector, string scalar, or function handle. To process the latest news stories, you can write your own custom event handler function. This function must have an input argument specified as a table. Each new news story from Money.Net is a single row in a table. For details about working with custom event handler functions, see “Writing and Running Custom Event Handler Functions” on page 1-26.

Specify this name-value pair argument only with the name-value pair argument `'Subscription'` and value `'on'`.

Example: `news(c,'Subscription','on','EventHandler',myFcn)`

Data Types: `char` | `function_handle` | `string`

Output Arguments

`n` — News stories
table

News stories, returned as a table with these variables. Each row in the table represents one news story. For details about these variables, see Money.Net API Documentation.

News Story Variable	Data Type	Variable Description
`ArticleTitle`	cell array of character vectors	News story title
`ArticleID`	double	Internal Money.Net identifier of the news story
`PublishedTime`	`datetime` array	Date and time the news story was published
`PublisherCode`	cell array of character vectors	Publisher four-digit code
`PublisherName`	cell array of character vectors	Publisher name
`ArticleBodyDescription`	cell array of character vectors	Body excerpt or short description of the news story
`Category`	cell array of character vectors	Internal Money.Net category of the news story
`URLLink`	cell array of character vectors	Website that contains the full news story
`Source`	cell array of character vectors	News stream source code

News Story Variable	Data Type	Variable Description
`Priority`	cell array of character vectors	Priority of news story with these values: • `0` is Normal • `1` is Breaking • `2` is Major Hot
`Symbols`	cell array	Symbols, or tickers, associated with the news story with these values: • An empty cell array for no symbols • A cell array that contains the symbol as a character vector for one symbol • A nested cell array that contains symbols as character vectors for multiple symbols

Version History

Introduced in R2016b

See Also

`moneynet` | `getdata` | `realtime` | `timeseries` | `close`

Topics

“Retrieve Money.Net News Stories” on page 8-7
“Writing and Running Custom Event Handler Functions” on page 1-26
“Money.Net Error and Warning Messages” on page 8-10

External Websites

Money.Net API Documentation

rnseloader

Retrieve data from Machine Readable News from Refinitiv sentiment archive file

Syntax

```
x = rnseloader(file)
x = rnseloader(file, 'date', {DATE1})
x = rnseloader(file, 'date', {DATE1, DATE2})
x = rnseloader(file, 'security', {SECNAME})
x = rnseloader(file, 'start', STARTREC)
x = rnseloader(file, 'records', NUMRECORDS)
x = rnseloader(file, 'fieldnames', F)
```

Arguments

Specify the following arguments as name-value pairs. You can specify any combination of name-value pairs in a single call to `rnseloader`.

`file`	Machine Readable News sentiment archive file from which to retrieve data.
`'date'`	Use this argument with `{DATE1,DATE2}` to retrieve data between and including the specified dates. Specify the dates as numbers, character vectors, or strings.
`'security'`	Use this argument to retrieve data for `SECNAME`, where `SECNAME` is a cell array containing a list of security identifiers for which to retrieve data.
`'start'`	Use this argument to retrieve data beginning with the record `STARTREC`, where `STARTREC` is the record at which `rnseloader` begins to retrieve data. Specify `STARTREC` as a number.
`'records'`	Use this argument to retrieve `NUMRECORDS` number of records.

Description

`x = rnseloader(file)` retrieves data from the Machine Readable News sentiment archive file `file`, and stores it in the structure `x`.

`x = rnseloader(file, 'date', {DATE1})` retrieves data from `file` with date stamps of value `DATE1`.

`x = rnseloader(file, 'date', {DATE1, DATE2})` retrieves data from `file` with date stamps between `DATE1` and `DATE2`.

`x = rnseloader(file, 'security', {SECNAME})` retrieves data from `file` for the securities specified by `SECNAME`.

`x = rnseloader(file, 'start', STARTREC)` retrieves data from `file` beginning with the record specified by `STARTREC`.

`x = rnseloader(file, 'records', NUMRECORDS)` retrieves `NUMRECORDS` number of records from `file`.

`x = rnseloader(file, 'fieldnames', F)` retrieves only the specified fields, `F`, in the output structure.

Examples

Retrieve data from the file `'file.csv'` with date stamps of `'02/02/2007'`:

```
x = rnseloader('file.csv','date',{'02/02/2007'})
```

Retrieve data from `'file.csv'` between and including `'02/02/2007'` and `'02/03/2007'`:

```
 x = rnseloader('file.csv','date',{'02/02/2007',...
'02/03/2007'})
```

Retrieve data from `'file.csv'` for the security `'XYZ.O'`:

```
 x = rnseloader('file.csv','security',{'XYZ.O'})
```

Retrieve the first 10000 records from `'file.csv'`:

```
 x = rnseloader('file.csv','records',10000)
```

Retrieve data from `'file.csv'`, starting at record 100000:

```
  x = rnseloader('file.csv','start',100000)
```

Retrieve up to 100000 records from `'file.csv'`, for the securities `'ABC.N'` and `'XYZ.O'`, with date stamps between and including the dates `'02/02/2007'` and `'02/03/2007'`:

```
x = rnseloader('file.csv','records',100000,...
               'date',{'02/02/2007','02/03/2007'},...
               'security',{'ABC.N','XYZ.O'})
```

Version History

Introduced in R2008b

See Also

tlkrs

SIX Financial Information connection

Description

The `tlkrs` function creates a `tlkrs` object. The `tlkrs` object represents a SIX Financial Information connection.

After you create a `tlkrs` object, you can use the object functions to retrieve current and intraday tick data.

Creation

Syntax

```
c = tlkrs(ci,ui,password)
```

Description

`c = tlkrs(ci,ui,password)` creates a connection to the SIX Financial Information data service and sets the ci, ui and password properties.

Properties

`ci` — Customer identifier

character vector | string scalar

Customer identifier, specified as a character vector or string scalar. For credentials, contact SIX Financial Information.

Example: `'US12345'`

Data Types: `char` | `string`

`ui` — User identifier

character vector | string scalar

User identifier, specified as a character vector or string scalar. For credentials, contact SIX Financial Information.

Example: `'userapid01'`

Data Types: `char` | `string`

`password` — Password

character vector | string scalar

Password, specified as a character vector or string scalar. For credentials, contact SIX Financial Information.

Example: `'userapid10'`

Data Types: `char` | `string`

`sessionid` — Session identifier
character vector

This property is read-only.

Session identifier, specified as a character vector. The `tlkrs` function determines the session identifier from the file specified by the login URL.

Example: `'463487494'`

Data Types: `char`

Object Functions

close	Close connection to SIX Financial Information
getdata	Current SIX Financial Information data
history	End of day SIX Financial Information data
timeseries	SIX Financial Information intraday tick data
isconnection	Determine if SIX Financial Information connection is valid
tkfieldtoid	SIX Financial Information field names to identification string
tkidtofield	SIX Financial Information identification string to field name

Examples

Connect to SIX Financial Information

Create a SIX Financial Information connection. Then, retrieve current data for instruments. The current data you see when completing this example can differ from the output data shown.

Create a SIX Financial Information connection with a customer identifier, user identifier, and password. `c` is a `tlkrs` object.

```
ci = 'US12345';
ui = 'userapid01';
password = 'userapid10';
c = tlkrs(ci,ui,password)
```

```
c =

  tlkrs with properties:

          ci: 'US12345'
          ui: 'userapid01'
    password: 'userapid10'
   sessionid: '463487494'
```

Convert the bid, ask, and last price fields to the identifiers `ids`.

```
f = {'Bid','Ask','Last'};
typ = 'market';
ids =  tkfieldtoid(c,f,typ);
```

Retrieve the current data for the specified securities `s`.

```
s = {'1758999,149,134','275027,148,184'};
d = getdata(c,s,ids);
```

Display the pricing data.

```
d.P.v
```

```
ans =

  6×1 cell array

    {'64.36'  }
    {'64.37'  }
    {0×0 double}
    {'1.26' }
    {'1.26' }
    {0×0 double}
```

Close the SIX Financial Information connection.

```
close(c)
```

Version History

Introduced in R2011b

See Also

close

Close connection to SIX Financial Information

Syntax

```
close(C)
```

Description

`close(C)` closes the connection `C` to SIX Financial Information.

Input Arguments

`C` — Connection

SIX Financial Information connection

Connection, specified as a SIX Financial Information connection.

Version History

Introduced in R2011b

See Also

`tlkrs`

getdata

Current SIX Financial Information data

Syntax

```
D = getdata(c,s,f)
```

Description

`D = getdata(c,s,f)` returns the data for the fields `f` for the security list `s`.

Examples

Retrieve Pricing Data for Securities

Retrieve SIX Financial Information pricing data for specified securities.

```
% Connect to Telekurs.
c = tlkrs('US12345','userapid01','userapid10')

% Convert specified fields to ID strings.
ids =  tkfieldtoid(c,{'Bid','Ask','Last'},'market');

% Retrieve data for specified securities.
d = getdata(c,{'1758999,149,134','275027,148,184'},ids);
```

Your output appears as follows:

```
d =
    XRF: [1x1 struct]
     IL: [1x1 struct]
      I: [1x1 struct]
      M: [1x1 struct]
      P: [1x1 struct]
```

`d.I` contains the instrument IDs, and `d.P` contains the pricing data.

View the instrument IDs like this:

```
d.I.k

ans =
    '1758999,149,134'
    '275027,148,184'
```

View the pricing data field IDs like this:

```
d.P.k

ans =

    '33,2,1'
```

```
    '33,3,1'
    '3,1,1'
    '33,2,1'
    '33,3,1'
    '3,1,1'
```

And the pricing data like this:

```
d.P.v

ans =

    '44.94'
    '44.95'
    []
    '0.9715'
    '0.9717'
    []
```

Convert field IDs in `d.P.k` to field names like this:

```
d.P.k = tkidtofield(c,d.P.k,'market')
```

Load the file `@tlkrs/tkfields.mat` for a listing of the field names (`Bid`, `Ask`, `Last`) and corresponding IDs.

Input Arguments

c — connection object

`tlkrs` object

Connection object, specified as a `tlkrs` object.

s — security

cell array of character vectors

Security, specified as a cell array of character vectors.

f — field IDs

cell array of character vectors

Field IDs, specified as a cell array of character vectors.

Version History

Introduced in R2011b

See Also

`tlkrs` | `history` | `timeseries` | `tkfieldtoid` | `tkidtofield`

history

End of day SIX Financial Information data

Syntax

```
D = history(c,s,f,fromdate,todate)
```

Description

`D = history(c,s,f,fromdate,todate)` returns the historical data for the security list `s`, for the fields `f`, for the dates `fromdate` to `todate`.

Examples

Retrieve End of Day Data for Security

Retrieve end of day SIX Financial Information data for the specified security for the past 5 days.

```
c = tlkrs('US12345','userapid01','userapid10')
ids = tkfieldtoid(c,{'Bid','Ask'},'history');
d = history(c,{'1758999,149,134'},ids,floor(now)-5,floor(now));
```

```
d =

    XRF: [1x1 struct]
     IL: [1x1 struct]
      I: [1x1 struct]
     HL: [1x1 struct]
     HD: [1x1 struct]
      P: [1x1 struct]
```

`d.I` contains the instrument IDs, `d.HD` contains the dates, and `d.P` contains the pricing data.

View the dates:

```
d.HD.d
```

```
ans =

    '20110225'
    '20110228'
    '20110301'
```

View the pricing field IDs:

```
d.P.k
```

```
ans =

    '3,2'
    '3,3'
    '3,2'
```

```
    '3,3'
    '3,2'
    '3,3'
```

View the pricing data:

```
d.P.v
```

```
ans =

    '45.32'
    '45.33'
    '45.26'
    '45.27'
    '44.94'
    '44.95'
```

Convert the field identification values in `d.P.k` to their corresponding field names like this:

```
d.P.k = tkidtofield(c,d.P.k,'history')
```

Input Arguments

`c` — connection object
`tlkrs` object

Connection object, specified as a `tlkrs` object.

`s` — security
cell array of character vectors

Security, specified as a cell array of character vectors.

`f` — field IDs
cell array of character vectors

Field IDs, specified as a cell array of character vectors.

`fromdate` — start date
serial date number

Start date, specified as a serial date number.

`todate` — end date
serial date number

End date, specified as a serial date number.

Version History

Introduced in R2011b

See Also

`tlkrs` | `getdata` | `timeseries` | `tkfieldtoid` | `tkidtofield`

isconnection

Determine if SIX Financial Information connection is valid

Syntax

```
X = isconnection(c)
```

Description

`X = isconnection(c)` returns `true` if `c` is a valid SIX Financial Information connection and `false` otherwise.

Input Arguments

c — connection object

`tlkrs` object

Connection object, specified as a `tlkrs` object.

Version History

Introduced in R2011b

See Also

`tlkrs` | `close` | `getdata`

timeseries

SIX Financial Information intraday tick data

Syntax

```
D = timeseries(c,s,t)
D = timeseries(c,s,{startdate,enddate})
D = timeseries(c,s,t,5)
```

Description

`D = timeseries(c,s,t)` returns the raw tick data for the SIX Financial Information connection object `c`, the security `s`, and the date `t`. Every trade, best, and ask tick is returned for the given date or date range.

`D = timeseries(c,s,{startdate,enddate})` returns the raw tick data for the security `s`, for the date range defined by `startdate` and `enddate`.

`D = timeseries(c,s,t,5)` returns the tick data for the security `s`, for the date `t` in intervals of 5 minutes, for the field `f`. Intraday tick data requested is returned in 5-minute intervals, with the columns representing:

- First
- High
- Low
- Last
- Volume weighted average
- Moving average

Examples

Retrieve Intraday Tick Data

Retrieve SIX Financial Information intraday tick data for the past 2 days:

```
c = tlkrs('US12345','userapid01','userapid10')
d = timeseries(c,{'1758999,149,134'}, ...
   {floor(now)-.25,floor(now)})
```

Display the returned data:

```
d =

    XRF: [1x1 struct]
     IL: [1x1 struct]
      I: [1x1 struct]
    TSL: [1x1 struct]
     TS: [1x1 struct]
      P: [1x1 struct]
```

`d.I` contains the instrument IDs, `d.TS` contains the date and time data, and `d.P` contains the pricing data.

Display the tick times:

```
d.TS.t(1:10)
```

```
ans =

    '013500'
    '013505'
    '013510'
    '013520'
    '013530'
    '013540'
    '013550'
    '013600'
    '013610'
    '013620'
```

Display the field IDs:

```
d.P.k(1:10)
```

```
ans =

    '3,4'
    '3,2'
    '3,3'
    '3,4'
    '3,2'
    '3,3'
    '3,4'
    '3,2'
    '3,3'
    '3,4'
```

Convert these IDs to field names (`Mid`, `Bid`, `Ask`) with `tkidtofield`:

```
d.P.k = tkidtofield(c,d.P.k,'history')
```

Load the file `@tlkrs/tkfields.mat` for a listing of the field names and corresponding IDs.

Display the corresponding tick values:

```
d.P.v(1:10)
```

```
ans =

    '45.325'
    '45.32'
    '45.33'
    '45.325'
    '45.32'
    '45.33'
    '45.325'
    '45.32'
```

```
    '45.33'
    '45.325'
```

Input Arguments

c — connection object
`tlkrs` object

Connection object, specified as a `tlkrs` object.

s — security
cell array of character vectors

Security, specified as a cell array of character vectors.

t — date
serial date number

Date, specified as a serial date number.

`startdate` — start date
serial date number

Start date, specified as a serial date number.

`enddate` — end date
serial date number

End date, specified as a serial date number.

Version History

Introduced in R2011b

See Also

`tlkrs` | `getdata` | `history`

tkfieldtoid

SIX Financial Information field names to identification string

Syntax

```
D = tkfieldtoid(c,f,typ)
```

Description

`D = tkfieldtoid(c,f,typ)` converts SIX Financial Information field names to their corresponding identification strings. `c` is the SIX Financial Information connection object, `f` is the field list, and `typ` denotes the field. Options for the field include market, `'market'`; time and sales, `'tass'`; and history, `'history'`. `market` fields are used with `getdata`, `tass` fields are used with `timeseries`, and `history` fields are used with `history`.

Examples

Retrieve Pricing Data

Retrieve pricing data associated with specified identification strings:

```
% Connect to SIX Telekurs.
c = tlkrs('US12345','userapid01','userapid10')

% Convert field names to identification strings.
ids = tkfieldtoid(c,{'bid','ask','last'},'market');

% Retrieve data associated with the identification strings.
d = getdata(c,{'1758999,149,134','275027,148,184',ids);
```

Input Arguments

c — connection object

`tlkrs` object

Connection object, specified as a `tlkrs` object.

f — field names

cell array of character vectors

Field names, specified as a cell array of character vectors.

typ — field

`'market'` | `'tass'` | `'history'`

Field, specified as `'market'`, `'tass'`, or `'history'`.

Version History

Introduced in R2011b

See Also

`tlkrs` | `getdata` | `history` | `timeseries` | `tkidtofield`

tkidtofield

SIX Financial Information identification string to field name

Syntax

```
D = tkidtofield(c,f,typ)
```

Description

`D = tkidtofield(c,f,typ)` converts SIX Financial Information field identification strings to their corresponding field names. `c` is the SIX Financial Information connection object, `f` is the ID list, and `typ` denotes the fields. Options for the fields include market, `'market'`; time and sales, `'tass'`; and history, `'history'`. `market` fields are used with `getdata`, `tass` fields are used with `timeseries`, and `history` fields are used with the `history`.

Examples

Convert Field Identification Strings to Field Names

When you retrieve output from SIX Financial Information, it appears as follows:

```
d =
    XRF: [1x1 struct]
     IL: [1x1 struct]
      I: [1x1 struct]
      M: [1x1 struct]
      P: [1x1 struct]
```

The instrument IDs are found in `d.I`, and the pricing data is found in `d.P`. The output for `d.P.k` appears like this:

```
ans =

    '33,2,1'
    '33,3,1'
    '3,1,1'
    '33,2,1'
    '33,3,1'
    '3,1,1'
```

Convert the field IDs in `d.P.k` to their field names with the `tkidtofield` function:

```
d.P.k = tkidtofield(c,d.P.k,'market')
```

Load the file `@tlkrs/tkfields.mat` for a listing of the field names and their corresponding field IDs.

Input Arguments

c — connection object
`tlkrs` object

Connection object, specified as a `tlkrs` object.

f — field IDs
cell array of character vectors

Field IDs, specified as a cell array of character vectors.

typ — field
`'market'` | `'tass'` | `'history'`

Field, specified as `'market'`, `'tass'`, or `'history'`.

Version History

Introduced in R2011b

See Also

`tlkrs` | `getdata` | `history` | `timeseries` | `tkfieldtoid`

twitter

Twitter connection object

Description

The `twitter` function creates a `twitter` object, which represents a Twitter connection.

To establish the connection, you must obtain these required credentials from Twitter:

- Consumer key
- Consumer secret
- Access token
- Access token secret

To obtain these credentials, you must first log in to your Twitter account. Then, fill out the form in Create an application.

After you create a `twitter` object, you can use the object functions to retrieve historical Twitter data with a Twitter connection. Or, you can retrieve other Twitter data by using the Twitter REST API to access other REST API endpoints.

Creation

Syntax

```
c = twitter(consumerkey,consumersecret,accesstoken,accesstokensecret)
```

Description

`c = twitter(consumerkey,consumersecret,accesstoken,accesstokensecret)` creates a Twitter connection using the consumer key, consumer secret, access token, and access token secret.

Input Arguments

`consumerkey` — Consumer key
character vector | string scalar

Consumer key (API key), specified as a character vector or string scalar. To obtain your consumer key, fill out the form in Create an application.

Data Types: `char` | `string`

`consumersecret` — Consumer secret
character vector | string scalar

Consumer secret (API secret), specified as a character vector or string scalar. To obtain your consumer secret, fill out the form in Create an application.

Data Types: `char` | `string`

accesstoken — Access token
character vector | string scalar

Access token, specified as a character vector or string scalar. To obtain your access token, fill out the form in Create an application.

Data Types: `char` | `string`

accesstokensecret — Access token secret
character vector | string scalar

Access token secret, specified as a character vector or string scalar. To obtain your access token secret, fill out the form in Create an application.

Data Types: `char` | `string`

Properties

Name — Account name
character vector

Account name in the Twitter profile, specified as a character vector.

Data Types: `char`

ScreenName — Twitter user name
character vector

Twitter user name, specified as a character vector.

Example: `@username`

Data Types: `char`

MetaData — Twitter account and profile information
structure

Twitter account and profile information, specified as a structure.

After creating a Twitter connection, you can access your account and profile information using the `MetaData` property. For example:

```
c.MetaData

ans =

  struct with fields:

                                  id: 1.234e+17
                              id_str: '123456789101112141'
                                name: 'Full Name'
                         screen_name: 'username'
                         ...
```

(The values here do not represent real Twitter data.)

Data Types: `struct`

`StatusCode` — Connection status code

`matlab.net.http.StatusCode` object

Connection status code, specified as a `matlab.net.http.StatusCode` object. When this property has the value `OK`, the Twitter connection is successful.

Object Functions

search	Search for Tweets
getdata	Retrieve Twitter data
postdata	Post Twitter data

Examples

Search for Tweets

Use a Twitter connection object to search for Tweets.

Create a Twitter connection using your credentials. (The values in this example do not represent real Twitter credentials.)

```
consumerkey = 'abcdefghijklmnop123456789';
consumersecret = 'qrstuvwxyz123456789';
accesstoken = '123456789abcdefghijklmnop';
accesstokensecret = '123456789qrstuvwxyz';

c = twitter(consumerkey,consumersecret,accesstoken,accesstokensecret);
```

Check the Twitter connection. If the `StatusCode` property has the value `OK`, the connection is successful.

```
c.StatusCode
```

```
ans =

    OK
```

Search for Tweets using the Twitter connection object and the search term `MathWorks`.

```
tweetquery = 'MathWorks';
d = search(c,tweetquery)
```

```
d =

  ResponseMessage with properties:

    StatusLine: 'HTTP/1.1 200 OK'
    StatusCode: OK
        Header: [1×25 matlab.net.http.HeaderField]
          Body: [1×1 matlab.net.http.MessageBody]
     Completed: 0
```

`d` is a `matlab.net.http.ResponseMessage` object. The `StatusCode` property shows `OK`, indicating a successful HTTP request.

Access MathWorks Tweets. Display the 12th Tweet.

```
 d.Body.Data.statuses{12}.text

ans =

    'MATLAB Control Systems Examples https://t.co/g2P86srv33'
```

You can search for other Tweets using the `search` function. To retrieve other Twitter data, use the `getdata` function.

Version History

Introduced in R2017b

See Also

`matlab.net.http.StatusCode` | `matlab.net.http.ResponseMessage`

Topics

“Conduct Sentiment Analysis Using Historical Tweets” on page 9-2
“Tweet Based on Retrieved Twitter Data” on page 9-6

External Websites

Create an application
Twitter REST API Endpoint Reference Documentation

search

Search for Tweets

Syntax

```
d = search(c,tweetquery)

d = search(c,tweetquery,parameters)
d = search(c,tweetquery,QueryName1,QueryValue1,...,QueryNameN,QueryValueN)
```

Description

`d = search(c,tweetquery)` searches Tweets for the term `tweetquery`.

`d = search(c,tweetquery,parameters)` searches Tweets using web service query parameters. The Twitter REST API defines web service query parameters.

`d = search(c,tweetquery,QueryName1,QueryValue1,...,QueryNameN,QueryValueN)` specifies web service query parameters as one or more pairs of name-value arguments.

Examples

Search for Tweets

Use a Twitter connection object to search for Tweets.

Create a Twitter connection using your credentials. (The values in this example do not represent real Twitter credentials.)

```
consumerkey = 'abcdefghijklmnop123456789';
consumersecret = 'qrstuvwxyz123456789';
accesstoken = '123456789abcdefghijklmnop';
accesstokensecret = '123456789qrstuvwxyz';

c = twitter(consumerkey,consumersecret,accesstoken,accesstokensecret);
```

Check the Twitter connection. If the `StatusCode` property has the value `OK`, the connection is successful.

```
c.StatusCode

ans =

    OK
```

Search for Tweets using the Twitter connection object and the search term `MathWorks`.

```
tweetquery = 'MathWorks';
d = search(c,tweetquery)

d =
```

```
  ResponseMessage with properties:

    StatusLine: 'HTTP/1.1 200 OK'
    StatusCode: OK
        Header: [1×25 matlab.net.http.HeaderField]
          Body: [1×1 matlab.net.http.MessageBody]
     Completed: 0
```

`d` is a `matlab.net.http.ResponseMessage` object. The `StatusCode` property shows `OK`, indicating a successful HTTP request.

Access MathWorks Tweets. Display the 12th Tweet.

```
d.Body.Data.statuses{12}.text
```

```
ans =

    'MATLAB Control Systems Examples https://t.co/g2P86srv33'
```

You can search for other Tweets using the `search` function. To retrieve other Twitter data, use the `getdata` function.

Search for Specific Number of Tweets Using Structure

Use a Twitter connection object to search for 20 Tweets. Specify the number of Tweets to retrieve as a structure.

Create a Twitter connection using your credentials. (The values in this example do not represent real Twitter credentials.)

```
consumerkey = 'abcdefghijklmnop123456789';
consumersecret = 'qrstuvwxyz123456789';
accesstoken = '123456789abcdefghijklmnop';
accesstokensecret = '123456789qrstuvwxyz';

c = twitter(consumerkey,consumersecret,accesstoken,accesstokensecret);
```

Check the Twitter connection. If the `StatusCode` property has the value `OK`, the connection is successful.

```
c.StatusCode
```

```
ans =

    OK
```

Specify the search term `MathWorks` in the variable `tweetquery`. Specify 20 Tweets as a field in the structure `parameters`. Search for 20 Tweets using the Twitter connection object, search term `tweetquery`, and structure `parameters`.

```
tweetquery = 'MathWorks';
parameters.count = 20;
d = search(c,tweetquery,parameters)
```

```
d =
```

```
  ResponseMessage with properties:

    StatusLine: 'HTTP/1.1 200 OK'
    StatusCode: OK
        Header: [1×25 matlab.net.http.HeaderField]
          Body: [1×1 matlab.net.http.MessageBody]
     Completed: 0
```

`d` is a `matlab.net.http.ResponseMessage` object. The `StatusCode` property shows `OK`, indicating a successful HTTP request.

Access MathWorks Tweets. Display the structure `Data`.

```
d.Body.Data
```

```
ans =

  struct with fields:

           statuses: {20×1 cell}
    search_metadata: [1×1 struct]
```

The structure `Data` contains the field `statuses`. This field is a cell array of structures. Each structure in the cell array contains information about one Tweet.

Access all 20 Tweets.

```
d.Body.Data.statuses{:}
```

```
ans =

  struct with fields:

                  created_at: 'Fri Apr 28 17:51:55 +0000 2017'
                          id: 1.2345e+17
                      id_str: '123456789101112131'
                        text: 'This collection of over 400 MATLAB examples can help you with #controlsystems, Kalman filters, and more
                   truncated: 0
                    entities: [1×1 struct]
                    metadata: [1×1 struct]
                      source: 'Twitter for iPhone'
       in_reply_to_status_id: []
    in_reply_to_status_id_str: []
          in_reply_to_user_id: []
      in_reply_to_user_id_str: []
      in_reply_to_screen_name: []
                         user: [1×1 struct]
                          geo: []
                  coordinates: []
                        place: []
                 contributors: []
            retweeted_status: [1×1 struct]
              is_quote_status: 0
                retweet_count: 34
               favorite_count: 0
                    favorited: 0
                    retweeted: 0
          possibly_sensitive: 0
                         lang: 'en'
...
```

The field `text` in each structure contains the text of one Tweet.

(These values do not represent real Twitter data.)

You can search for other Tweets using the `search` function. To retrieve other Twitter data, use the `getdata` function.

Search for Specific Number of Tweets Using Name-Value Arguments

Use a Twitter connection object to search for 20 Tweets. Specify the number of Tweets to retrieve as a name-value argument.

Create a Twitter connection using your credentials. (The values in this example do not represent real Twitter credentials.)

```
consumerkey = 'abcdefghijklmnop123456789';
consumersecret = 'qrstuvwxyz123456789';
accesstoken = '123456789abcdefghijklmnop';
accesstokensecret = '123456789qrstuvwxyz';

c = twitter(consumerkey,consumersecret,accesstoken,accesstokensecret);
```

Check the Twitter connection. If the `StatusCode` property has the value `OK`, the connection is successful.

```
c.StatusCode
```

```
ans =

    OK
```

Search for 20 Tweets using the Twitter connection object, search term `MathWorks`, and name-value argument `count`.

```
tweetquery = 'MathWorks';
d = search(c,tweetquery,'count',20)
```

```
d =

  ResponseMessage with properties:

    StatusLine: 'HTTP/1.1 200 OK'
    StatusCode: OK
        Header: [1×25 matlab.net.http.HeaderField]
          Body: [1×1 matlab.net.http.MessageBody]
     Completed: 0
```

`d` is a `matlab.net.http.ResponseMessage` object. The `StatusCode` property shows `OK`, indicating a successful HTTP request.

Access MathWorks Tweets. Display the structure `Data`.

```
 d.Body.Data
```

```
ans =

  struct with fields:

           statuses: {20×1 cell}
    search_metadata: [1×1 struct]
```

The structure `Data` contains the field `statuses`. This field is a cell array of structures. Each structure in the cell array contains information about one Tweet.

Access all 20 Tweets.

```
d.Body.Data.statuses{:}
```

```
ans =

  struct with fields:

                  created_at: 'Fri Apr 28 17:51:55 +0000 2017'
                          id: 1.2345e+17
                      id_str: '123456789101112131'
                        text: 'This collection of over 400 MATLAB examples can help you with #controlsystems, Kalman filters, and more
                   truncated: 0
                    entities: [1×1 struct]
                    metadata: [1×1 struct]
                      source: 'Twitter for iPhone'
       in_reply_to_status_id: []
   in_reply_to_status_id_str: []
         in_reply_to_user_id: []
     in_reply_to_user_id_str: []
     in_reply_to_screen_name: []
                        user: [1×1 struct]
                         geo: []
                coordinates: []
                       place: []
                contributors: []
            retweeted_status: [1×1 struct]
             is_quote_status: 0
               retweet_count: 34
              favorite_count: 0
                   favorited: 0
                   retweeted: 0
          possibly_sensitive: 0
                        lang: 'en'
...
```

The field `text` in each structure contains the text of one Tweet.

(These values do not represent real Twitter data.)

You can search for other Tweets using the `search` function. To retrieve other Twitter data, use the `getdata` function.

Input Arguments

`c` — Twitter connection

`twitter` object

Twitter connection, specified as a `twitter` object.

`tweetquery` — Tweet search term

character vector | string scalar

Tweet search term, specified as a character vector or string scalar.

Example: `"MathWorks"`

Data Types: `char` | `string`

`parameters` — Web service query parameters

structure

Web service query parameters, specified as a structure. Each parameter is specified as a field in the structure. Set the field to a specific value in the structure. For example, specify the number of Tweets to retrieve:

```
parameters.count = 20;
```

The Twitter REST API defines web service query parameters that it accepts as part of an HTTP request.

Data Types: `struct`

`QueryName1,QueryValue1,...,QueryNameN,QueryValueN` — Web service query parameters
name-value pairs

Web service query parameters, specified as one or more pairs of name-value arguments. A `QueryName` argument is a character vector or string scalar that specifies the name of a query parameter. A `QueryValue` argument is a character vector or string scalar that specifies the value of the query parameter.

The Twitter REST API defines web service query parameters that it accepts as part of an HTTP request.

Example: `'count',20` retrieves 20 Tweets.

Data Types: `char` | `string`

Output Arguments

`d` — Twitter data
`matlab.net.http.ResponseMessage`

Twitter data, returned as a `matlab.net.http.ResponseMessage` object.

To retrieve Twitter data, access properties in `d`, for example:

```
data = d.Body.Data
```

```
data =

      struct with fields:

               statuses: {50×1 cell}
        search_metadata: [1×1 struct]
```

Continue to access the nested structure `data` to retrieve Twitter data. For accessing nested structures, see “Access Data in Nested Structures”.

Limitations

- The Twitter REST API `GET search/tweets` endpoint specifies that you can retrieve up to a maximum of 100 Tweets at a time.
- The Twitter REST API `GET search/tweets` endpoint specifies that you can retrieve up to 7 days of historical Tweets.

Version History

Introduced in R2017b

See Also

Functions
`getdata` | `postdata`

Objects
`twitter`

Topics
“Conduct Sentiment Analysis Using Historical Tweets” on page 9-2

External Websites
Twitter REST API Endpoint Reference Documentation

getdata

Retrieve Twitter data

Syntax

```
d = getdata(c,baseurl)
d = getdata(c,baseurl,parameters)
d = getdata(c,baseurl,QueryName1,QueryValue1,...,QueryNameN,QueryValueN)
```

Description

`d = getdata(c,baseurl)` retrieves Twitter data for REST API GET endpoints that do not require any web service query parameters.

`d = getdata(c,baseurl,parameters)` retrieves Twitter data using web service query parameters. The Twitter REST API defines web service query parameters for each endpoint. For valid parameters, see the Twitter REST API Endpoint Reference Documentation.

`d = getdata(c,baseurl,QueryName1,QueryValue1,...,QueryNameN,QueryValueN)` specifies web service query parameters as one or more pairs of name-value arguments.

Examples

Retrieve Twitter Data Without Specifying Parameters

Use a Twitter connection object to return locations for trending topics. The REST API endpoint `GET trends/available` does not require any web service query parameters.

Create a Twitter connection using your credentials. (The values in this example do not represent real Twitter credentials.)

```
consumerkey = 'abcdefghijklmnop123456789';
consumersecret = 'qrstuvwxyz123456789';
accesstoken = '123456789abcdefghijklmnop';
accesstokensecret = '123456789qrstuvwxyz';

c = twitter(consumerkey,consumersecret,accesstoken,accesstokensecret);
```

Check the Twitter connection. If the `StatusCode` property has the value `OK`, the connection is successful.

```
c.StatusCode

ans =

    OK
```

Specify the Twitter base URL.

```
baseurl = 'https://api.twitter.com/1.1/trends/available.json';
```

Retrieve locations for trending topics using the Twitter connection object and the base URL.

```
d = getdata(c,baseurl)
```

```
d =

  ResponseMessage with properties:

    StatusLine: 'HTTP/1.1 200 OK'
    StatusCode: OK
        Header: [1×25 matlab.net.http.HeaderField]
          Body: [1×1 matlab.net.http.MessageBody]
     Completed: 0
```

`d` is a `matlab.net.http.ResponseMessage` object. The `StatusCode` property shows `OK`, indicating a successful HTTP request.

Access the location data. Display the structure `Data`.

```
d.Body.Data
```

```
ans =

  467×1 struct array with fields:

    name
    placeType
    url
    parentid
    country
    woeid
    countryCode
```

The structure `Data` is a structure array with the field `name`, which contains the name of a location for a trending topic.

Access the first location.

```
d.Body.Data(1).name
```

```
ans =

    'Worldwide'
```

You can retrieve data for other REST API endpoints by substituting another URL for the `baseurl` input argument. Or, you can search for Tweets using the `search` function.

Retrieve Twitter Data Using Structure

Use a Twitter connection object to retrieve follower information. Specify the count of followers as a structure.

Create a Twitter connection using your credentials. (The values in this example do not represent real Twitter credentials.)

```
consumerkey = 'abcdefghijklmnop123456789';
consumersecret = 'qrstuvwxyz123456789';
```

```
accesstoken = '123456789abcdefghijklmnop';
accesstokensecret = '123456789qrstuvwxyz';

c = twitter(consumerkey,consumersecret,accesstoken,accesstokensecret);
```

Check the Twitter connection. If the `StatusCode` property has the value `OK`, the connection is successful.

```
c.StatusCode
```

```
ans =

    OK
```

Set the Twitter base URL to access the `GET followers/list` REST API endpoint. Specify one follower by defining the structure `parameters` with the field set to 1. Search for one follower of the current account using the Twitter connection object, base URL, and structure `parameters`.

```
baseurl = 'https://api.twitter.com/1.1/followers/list.json';
parameters.count = 1;
d = getdata(c,baseurl,parameters)
```

```
d =

  ResponseMessage with properties:

    StatusLine: 'HTTP/1.1 200 OK'
    StatusCode: OK
        Header: [1×25 matlab.net.http.HeaderField]
          Body: [1×1 matlab.net.http.MessageBody]
     Completed: 0
```

`d` is a `matlab.net.http.ResponseMessage` object. The `StatusCode` property shows `OK`, indicating a successful HTTP request.

Access information about the follower.

```
d.Body.Data.users
```

```
ans =

  struct with fields:

                                    id: 12345678
                                id_str: '12345678'
                                  name: 'Full Name'
...
```

`d.Body.Data.users` is a structure that has a field for each piece of account information. For example, the first three fields are:

- Account identifier as a number
- Account identifier as a character vector
- Full name of the account as a character vector

(These values do not represent real Twitter data.)

You can retrieve data for other REST API endpoints by substituting another URL for the `baseurl` input argument. Or, you can search for Tweets using the `search` function.

Retrieve Twitter Data Using Name-Value Arguments

Use a Twitter connection object to retrieve follower information. Specify the count of followers as a name-value argument.

Create a Twitter connection using your credentials. (The values in this example do not represent real Twitter credentials.)

```
consumerkey = 'abcdefghijklmnop123456789';
consumersecret = 'qrstuvwxyz123456789';
accesstoken = '123456789abcdefghijklmnop';
accesstokensecret = '123456789qrstuvwxyz';

c = twitter(consumerkey,consumersecret,accesstoken,accesstokensecret);
```

Check the Twitter connection. If the `StatusCode` property has the value `OK`, the connection is successful.

```
c.StatusCode
```

```
ans =

    OK
```

Set the Twitter base URL to access the `GET followers/list` REST API endpoint. Search for one follower of the current account using the Twitter connection object, base URL, and name-value argument `count`.

```
baseurl = 'https://api.twitter.com/1.1/followers/list.json';
d = getdata(c,baseurl,'count',1)
```

```
d =

  ResponseMessage with properties:

    StatusLine: 'HTTP/1.1 200 OK'
    StatusCode: OK
        Header: [1×25 matlab.net.http.HeaderField]
          Body: [1×1 matlab.net.http.MessageBody]
     Completed: 0
```

`d` is a `matlab.net.http.ResponseMessage` object. The `StatusCode` property shows `OK`, indicating a successful HTTP request.

Access information about the follower.

```
d.Body.Data.users
```

```
ans =

  struct with fields:
```

```
        id: 12345678
    id_str: '12345678'
      name: 'Full Name'
...
```

`d.Body.Data.users` is a structure that has a field for each piece of account information. For example, the first three fields are:

- Account identifier as a number
- Account identifier as a character vector
- Full name of the account as a character vector

(These values do not represent real Twitter data.)

You can retrieve data for other REST API endpoints by substituting another URL for the `baseurl` input argument. Or, you can search for Tweets using the `search` function.

Input Arguments

`c` — Twitter connection

`twitter` object

Twitter connection, specified as a `twitter` object.

`baseurl` — Twitter base URL

character vector | string scalar

Twitter base URL, specified as a character vector or string scalar. Use this URL to access the Twitter REST API endpoints.

Example: `'https://api.twitter.com/1.1/followers/list.json'` specifies a GET REST API endpoint.

Data Types: `char` | `string`

`parameters` — Web service query parameters

structure

Web service query parameters, specified as a structure. Each parameter is specified as a field in the structure. Set the field to a specific value in the structure. For example, specify the number of items for the HTTP request:

```
parameters.count = 20;
```

The Twitter REST API defines web service query parameters that it accepts as part of an HTTP request. For valid parameters, see the Twitter REST API Endpoint Reference Documentation.

Data Types: `struct`

`QueryName1,QueryValue1,...,QueryNameN,QueryValueN` — Web service query parameters

name-value pairs

Web service query parameters, specified as one or more pairs of name-value arguments. A `QueryName` argument is a character vector or string scalar that specifies the name of a query parameter. A `QueryValue` argument is a character vector or string scalar that specifies the value of the query parameter.

The Twitter REST API defines web service query parameters that it accepts as part of an HTTP request. For valid parameters, see the Twitter REST API Endpoint Reference Documentation.

Example: `'count',20` specifies the number of items for the HTTP request.

Data Types: `char` | `string`

Output Arguments

d — Twitter data

`matlab.net.http.ResponseMessage`

Twitter data, returned as a `matlab.net.http.ResponseMessage` object.

To retrieve Twitter data, access properties in `d`, for example:

```
data = d.Body.Data

data =

      struct with fields:

               statuses: {50×1 cell}
        search_metadata: [1×1 struct]
```

Continue to access the nested structure `data` to retrieve Twitter data. For accessing nested structures, see “Access Data in Nested Structures”.

Limitations

- Each Twitter REST GET API endpoint has its own limitations. For details, see the Twitter REST API Endpoint Reference Documentation.

Version History

Introduced in R2017b

See Also

Functions

`search` | `postdata`

Objects

`twitter`

Topics

“Tweet Based on Retrieved Twitter Data” on page 9-6

External Websites

Twitter REST API Endpoint Reference Documentation

postdata

Post Twitter data

Syntax

```
d = postdata(c,baseurl)
d = postdata(c,baseurl,parameters)
d = postdata(c,baseurl,QueryName1,QueryValue1,...,QueryNameN,QueryValueN)
```

Description

`d = postdata(c,baseurl)` posts Twitter data for REST API POST endpoints that do not require any web service query parameters.

`d = postdata(c,baseurl,parameters)` posts Twitter data using web service query parameters. The Twitter REST API defines web service query parameters for each endpoint. For valid parameters, see the Twitter REST API Endpoint Reference Documentation.

`d = postdata(c,baseurl,QueryName1,QueryValue1,...,QueryNameN,QueryValueN)` specifies web service query parameters as one or more pairs of name-value arguments.

Examples

Post Twitter Data Without Specifying Parameters

Use a Twitter connection object to check Twitter account settings. The REST API endpoint `POST account/settings` does not require any web service query parameters.

Create a Twitter connection using your credentials. (The values in this example do not represent real Twitter credentials.)

```
consumerkey = 'abcdefghijklmnop123456789';
consumersecret = 'qrstuvwxyz123456789';
accesstoken = '123456789abcdefghijklmnop';
accesstokensecret = '123456789qrstuvwxyz';

c = twitter(consumerkey,consumersecret,accesstoken,accesstokensecret);
```

Check the Twitter connection. If the `StatusCode` property has the value `OK`, the connection is successful.

```
c.StatusCode
```

```
ans =

    OK
```

Specify the Twitter base URL.

```
baseurl = 'https://api.twitter.com/1.1/account/settings.json';
```

Retrieve account settings using the Twitter connection object and base URL.

```
d = postdata(c,baseurl)
```

```
d =

  ResponseMessage with properties:

    StatusLine: 'HTTP/1.1 200 OK'
    StatusCode: OK
        Header: [1×22 matlab.net.http.HeaderField]
          Body: [1×1 matlab.net.http.MessageBody]
     Completed: 0
```

`d` is a `matlab.net.http.ResponseMessage` object. The `StatusCode` property shows `OK`, indicating a successful HTTP request.

Access account settings data. Display the structure `Data`.

```
d.Body.Data
```

```
ans =

  struct with fields:

                      protected: 0
                    screen_name: 'screenName'
                always_use_https: 1
      use_cookie_personalization: 0
                     sleep_time: [1×1 struct]
                    geo_enabled: 0
                       language: 'en'
           discoverable_by_email: 0
    discoverable_by_mobile_phone: 0
         display_sensitive_media: 0
       allow_contributor_request: 'none'
                allow_dms_from: 'following'
           allow_dm_groups_from: 'following'
                translator_type: 'none'
```

(These values do not represent real Twitter data.)

You can post data using other REST API endpoints by substituting another URL for the `baseurl` input argument.

Post Twitter Data Using Structure

Use a Twitter connection object to create a Twitter search. Specify the search term for the saved search using a structure.

Create a Twitter connection using your credentials. (The values in this example do not represent real Twitter credentials.)

```
consumerkey = 'abcdefghijklmnop123456789';
consumersecret = 'qrstuvwxyz123456789';
accesstoken = '123456789abcdefghijklmnop';
```

```
accesstokensecret = '123456789qrstuvwxyz';

c = twitter(consumerkey,consumersecret,accesstoken,accesstokensecret);
```

Check the Twitter connection. If the `StatusCode` property has the value `OK`, the connection is successful.

```
c.StatusCode
```

```
ans =

    OK
```

Specify the search term `MathWorks` as a field of the structure `parameters`. Specify the Twitter base URL for the REST API POST endpoint `POST saved_searches/create`.

```
parameters.query = 'MathWorks';
baseurl = 'https://api.twitter.com/1.1/saved_searches/create.json';
```

Create a saved search using the Twitter connection object, base URL, and structure `parameters`.

```
d = postdata(c,baseurl,parameters)
```

```
d =

  ResponseMessage with properties:

    StatusLine: 'HTTP/1.1 200 OK'
    StatusCode: OK
        Header: [1×23 matlab.net.http.HeaderField]
          Body: [1×1 matlab.net.http.MessageBody]
     Completed: 0
```

`d` is a `matlab.net.http.ResponseMessage` object. The `StatusCode` property shows `OK`, indicating a successful HTTP request.

Access the saved search data.

```
d.Body.Data
```

```
ans =

  struct with fields:

              id: 8.6011e+17
       id_str: '860112019273416704'
        query: 'MathWorks'
         name: 'MathWorks'
     position: []
    created_at: 'Thu May 04 12:41:00 +0000 2017'
```

`d.Body.Data` is a structure that contains information about the saved search in fields. For example, the field `query` contains the search term `MathWorks` as a character vector.

You can post data using other REST API endpoints by substituting another URL for the `baseurl` input argument.

Post Twitter Data Using Name-Value Arguments

Use a Twitter connection object to create a Twitter search. Specify the search term for the saved search as a name-value argument.

Create a Twitter connection using your credentials. (The values in this example do not represent real Twitter credentials.)

```
consumerkey = 'abcdefghijklmnop123456789';
consumersecret = 'qrstuvwxyz123456789';
accesstoken = '123456789abcdefghijklmnop';
accesstokensecret = '123456789qrstuvwxyz';

c = twitter(consumerkey,consumersecret,accesstoken,accesstokensecret);
```

Check the Twitter connection. If the `StatusCode` property has the value `OK`, the connection is successful.

```
c.StatusCode
```

```
ans =

    OK
```

Specify the Twitter base URL for the REST API POST endpoint `POST saved_searches/create`.

```
baseurl = 'https://api.twitter.com/1.1/saved_searches/create.json';
```

Create a saved search for the search term `MathWorks` using the Twitter connection object, base URL, and name-value argument `query`.

```
d = postdata(c,baseurl,'query','MathWorks')
```

```
d =

  ResponseMessage with properties:

    StatusLine: 'HTTP/1.1 200 OK'
    StatusCode: OK
        Header: [1×23 matlab.net.http.HeaderField]
          Body: [1×1 matlab.net.http.MessageBody]
     Completed: 0
```

`d` is a `matlab.net.http.ResponseMessage` object. The `StatusCode` property shows `OK`, indicating a successful HTTP request.

Access the saved search data.

```
d.Body.Data
```

```
ans =

  struct with fields:

               id: 8.6011e+17
           id_str: '860112019273416704'
            query: 'MathWorks'
```

```
          name: 'MathWorks'
      position: []
    created_at: 'Thu May 04 12:41:00 +0000 2017'
```

`d.Body.Data` is a structure that contains information about the saved search in fields. For example, the field `query` contains the search term `MathWorks` as a character vector.

You can post data using other REST API endpoints by substituting another URL for the `baseurl` input argument.

Input Arguments

`c` — Twitter connection

`twitter` object

Twitter connection, specified as a `twitter` object.

`baseurl` — Twitter base URL

character vector | string scalar

Twitter base URL, specified as a character vector or string scalar. Use this URL to access the Twitter REST API endpoints.

Example: `'https://api.twitter.com/1.1/followers/list.json'` specifies a GET REST API endpoint.

Data Types: `char` | `string`

`parameters` — Web service query parameters

structure

Web service query parameters, specified as a structure. Each parameter is specified as a field in the structure. Set the field to a specific value in the structure. For example, specify the number of items for the HTTP request:

```
parameters.count = 20;
```

The Twitter REST API defines web service query parameters that it accepts as part of an HTTP request. For valid parameters, see the Twitter REST API Endpoint Reference Documentation.

Data Types: `struct`

`QueryName1,QueryValue1,...,QueryNameN,QueryValueN` — Web service query parameters

name-value pairs

Web service query parameters, specified as one or more pairs of name-value arguments. A `QueryName` argument is a character vector or string scalar that specifies the name of a query parameter. A `QueryValue` argument is a character vector or string scalar that specifies the value of the query parameter.

The Twitter REST API defines web service query parameters that it accepts as part of an HTTP request. For valid parameters, see the Twitter REST API Endpoint Reference Documentation.

Example: `'count',20` specifies the number of items for the HTTP request.

Data Types: `char` | `string`

Output Arguments

d — Twitter data
`matlab.net.http.ResponseMessage`

Twitter data, returned as a `matlab.net.http.ResponseMessage` object.

To retrieve Twitter data, access properties in `d`, for example:

```
data = d.Body.Data
```

```
data =

      struct with fields:

               statuses: {50×1 cell}
         search_metadata: [1×1 struct]
```

Continue to access the nested structure `data` to retrieve Twitter data. For accessing nested structures, see “Access Data in Nested Structures”.

Limitations

- Each Twitter REST POST API endpoint has its own limitations. For details, see the Twitter REST API Endpoint Reference Documentation.

Version History

Introduced in R2017b

See Also

Functions
`search` | `getdata`

Objects
`twitter`

Topics
“Tweet Based on Retrieved Twitter Data” on page 9-6

External Websites
Twitter REST API Endpoint Reference Documentation

trth

Tick History from Refinitiv connection

Description

The `trth` function creates a `trth` object, which represents a Tick History from Refinitiv connection. This connection uses the Tick History REST API to retrieve data. After you create a `trth` object, you can use the object functions to retrieve historical and intraday data.

Creation

Syntax

```
c = trth(username,password)
```

Description

`c = trth(username,password)` creates a Tick History from Refinitiv connection object using a password and sets the Username property.

Input Arguments

`password` — Password
character vector | string scalar

Password, specified as a character vector or string scalar. For credentials, contact Refinitiv.

Example: `'password'`

Data Types: `char` | `string`

Properties

`Username` — User name
character vector | string scalar

User name, specified as a character vector or string scalar. For credentials, contact Refinitiv.

Example: `'username'`

Data Types: `char` | `string`

`TimeOut` — Timeout
`200` (default) | numeric scalar

Timeout value in seconds that MATLAB attempts to connect by using the Tick History REST API, specified as a numeric scalar.

Example: `100`

Data Types: `double`

Object Functions

history	Tick History from Refinitiv historical data
timeseries	Tick History from Refinitiv intraday data

Examples

Retrieve Historical Data for One Security

Use a Tick History connection to retrieve historical data for a security.

Create a Tick History connection by using a user name and password. The appearance of the connection object `c` in the MATLAB workspace indicates a successful connection.

```
username = 'username';
password = 'password';
c = trth(username,password);
```

Retrieve historical data for the IBM security. Using the `history` function, retrieve the open and closing prices for the prior day.

```
sec = ["IBM.N","Ric"];
fields = ["Open";"Last"];
startdate = datetime('yesterday');
enddate = datetime('today');

d = history(c,sec,fields,startdate,enddate)
```

```
d =

  1×2 timetable

      Time         Open      Last
    ___________    ______    ______

    2017/11/02     154.25    153.35
```

`d` is a timetable that contains these variables:

- Date for the prior day
- Open price
- Closing price

Tips

- To access the latest Refinitiv API with the `trth` function, use Datafeed Toolbox R2022a or later.

Version History

Introduced in R2018a

See Also

Topics
“Decide to Buy Shares with Intraday Data Using Tick History from Refinitiv” on page 10-2
“Decide to Sell Shares with Historical Data Using Tick History from Refinitiv” on page 10-4

External Websites
Refinitiv REST API Documentation

history

Tick History from Refinitiv historical data

Syntax

```
d = history(c,sec,fields,startdate,enddate)
```

Description

`d = history(c,sec,fields,startdate,enddate)` returns historical data using:

- Tick History from Refinitiv connection object
- Refinitiv securities (for example, Reuters Instrument Codes, or RICs)
- Refinitiv historical fields
- Start date for the beginning of the historical date range
- End date for the end of the historical date range

Examples

Retrieve Historical Data for One Security

Use a Tick History connection to retrieve historical data for a security.

Create a Tick History connection by using a user name and password. The appearance of the connection object `c` in the MATLAB workspace indicates a successful connection.

```
username = 'username';
password = 'password';
c = trth(username,password);
```

Retrieve historical data for the IBM security. Using the `history` function, retrieve the open and closing prices for the prior day.

```
sec = ["IBM.N","Ric"];
fields = ["Open";"Last"];
startdate = datetime('yesterday');
enddate = datetime('today');

d = history(c,sec,fields,startdate,enddate)
```

```
d =

  1×2 timetable

      Time         Open      Last
    ___________    ______    ______

    2017/11/02     154.25    153.35
```

`d` is a timetable that contains these variables:

- Date for the prior day
- Open price
- Closing price

Retrieve Historical Data for Two Securities

Use a Tick History connection to retrieve historical data for two securities.

Create a Tick History from Refinitiv connection by using a user name and password. The appearance of the connection object `c` in the MATLAB workspace indicates a successful connection.

```
username = 'username';
password = 'password';
c = trth(username,password);
```

Retrieve historical data for the IBM and Ford Motor Company securities. Using the `history` function, retrieve the open and closing prices for the days in the date range from October 30, 2017, through November 3, 2017.

```
sec = ["IBM.N","Ric";"F.N","Ric"];
fields = ["Open";"Last"];
startdate = datetime('10/30/2017','InputFormat','MM/dd/yyyy');
enddate = datetime('11/03/2017','InputFormat','MM/dd/yyyy');

d = history(c,sec,fields,startdate,enddate)
```

```
d =

  10×2 timetable

      Time        Open      Last
    __________    ______    ______

    2017/10/30    153.76    154.36
    2017/10/31    154.26    154.06
    2017/11/01    153.97    154.03
    2017/11/02    154.25    153.35
    2017/11/03    153.36    151.58
    2017/10/30     12.00     12.10
    2017/10/31     12.14     12.27
    2017/11/01     12.40     12.35
    2017/11/02     12.33     12.42
    2017/11/03     12.40     12.36
```

`d` is a timetable that contains these variables:

- Date in the date range
- Open price
- Closing price

The first five rows contain data for the IBM security. The next five rows contain data for the Ford Motor Company security.

Input Arguments

`c` — Tick History from Refinitiv connection

connection object

Tick History from Refinitiv connection, specified as a connection object created with `trth`.

`sec` — Security

string array | cell array of character vectors

Security, specified as an N-by-2 string array or cell array of character vectors. The first column of the string array or cell array identifies the security. The second column identifies the type of security (for example, Reuters Instrument Code, or RIC).

Example: `["IBM.N","Ric"]`

Data Types: `string` | `cell`

`fields` — Fields

string array | cell array of character vectors

Fields, specified as a string array or cell array of character vectors. Specify Refinitiv historical fields to retrieve historical data. You can search for fields in MyRefinitiv.

Example: `["Low";"Last";"Volume"]`

Data Types: `string` | `cell`

`startdate` — Start date

`datetime` array | string scalar | character vector | numeric scalar

Start date of the date range, specified as a `datetime` array, string scalar, character vector, or numeric scalar.

Example: `datetime('yesterday')`

Data Types: `double` | `char` | `string` | `datetime`

`enddate` — End date

`datetime` array | string scalar | character vector | numeric scalar

End date of the date range, specified as a `datetime` array, string scalar, character vector, or numeric scalar.

Example: `datetime('today')`

Data Types: `double` | `char` | `string` | `datetime`

Output Arguments

`d` — Historical data

timetable

Historical data from Tick History, returned as a timetable. Except for the `Time` variable, each variable in the timetable corresponds to a specified field in the `fields` input argument.

Version History

Introduced in R2018a

See Also

`trth` | `timeseries`

Topics

“Decide to Sell Shares with Historical Data Using Tick History from Refinitiv” on page 10-4

External Websites

Refinitiv REST API Documentation

timeseries

Tick History from Refinitiv intraday data

Syntax

```
d = timeseries(c,sec,fields,startdate,enddate)
d = timeseries(c,sec,fields,startdate,enddate,interval)
```

Description

`d = timeseries(c,sec,fields,startdate,enddate)` returns intraday data using:

- Tick History from Refinitiv connection object
- Refinitiv securities (for example, Reuters Instrument Codes, or RICs)
- Refinitiv intraday fields
- Start date for the beginning of the intraday date range
- End date for the end of the intraday date range

`d = timeseries(c,sec,fields,startdate,enddate,interval)` uses an aggregation value for the intraday data.

Examples

Retrieve Intraday Data for One Security

Use a Tick History connection to retrieve intraday data for one security.

Create a Tick History from Refinitiv connection by using a user name and password. The appearance of the connection object `c` in the MATLAB workspace indicates a successful connection.

```
username = 'username';
password = 'password';
c = trth(username,password);
```

Retrieve intraday data for the IBM security. Using the `timeseries` function, retrieve the exchange time, price, and volume from November 6, 2017, through November 7, 2017.

```
sec = ["IBM.N","Ric"];
fields = ["Trade - Exchange Time";"Trade - Price";"Trade - Volume"];
startdate = datetime('11/06/2017','InputFormat','MM/dd/yyyy');
enddate = datetime('11/07/2017','InputFormat','MM/dd/yyyy');

d = timeseries(c,sec,fields,startdate,enddate);
```

Display the first three rows of intraday data.

```
head(d,3)
```

```
ans =
```

```
3×7 timetable

          Time             x_RIC         Domain        GMTOffset     Type       Price      Volu
    ____________________    _____    ______________    _________    _______    ________    ____

    06-Nov-2017 05:31:02    'IBM.N'    'Market Price'      '-5'       'Trade'    ''           Na
    06-Nov-2017 14:30:10    'IBM.N'    'Market Price'      '-5'       'Trade'    '151.68'    698
    06-Nov-2017 14:30:10    'IBM.N'    'Market Price'      '-5'       'Trade'    '151.66'
```

`d` is a timetable that contains these variables:

- Transaction date and time
- RIC
- Domain
- GMT time zone offset
- Transaction type
- Price
- Volume
- Exchange time

Retrieve Aggregated Intraday Data for Two Securities

Use a Tick History connection to retrieve intraday data for two securities, aggregated into 1-hour intervals.

Create a Tick History from Refinitiv connection by using a user name and password. The appearance of the connection object `c` in the MATLAB workspace indicates a successful connection.

```
username = 'username';
password = 'password';
c = trth(username,password);
```

Retrieve intraday data for the IBM and Ford Motor Company securities. Using the `timeseries` function, retrieve the open, high, low, and last prices from November 6, 2017, through November 7, 2017. Aggregate the intraday data into 1-hour intervals.

```
sec = ["IBM.N","Ric";"F.N","Ric"];
fields = ["Open";"High";"Low";"Last"];
startdate = datetime('11/06/2017','InputFormat','MM/dd/yyyy');
enddate = datetime('11/07/2017','InputFormat','MM/dd/yyyy');
interval = 'OneHour';

d = timeseries(c,sec,fields,startdate,enddate,interval);
```

`d` is a timetable that contains 66 rows of aggregated intraday data. The first 33 rows contain data for the Ford Motor Company security. The last 33 rows contain data for the IBM security.

Display three rows of Ford Motor Company aggregated intraday data.

```
d(15:17,:)

ans =
```

```
  3×8 timetable

          Time            x_RIC        Domain        GMTOffset          Type           Open
    ____________________    _____    _____________    _________    _________________    _____

    06-Nov-2017 14:00:00    'F.N'    'Market Price'      '-5'       'Intraday 1Hour'    12.34
    06-Nov-2017 15:00:00    'F.N'    'Market Price'      '-5'       'Intraday 1Hour'    12.38
    06-Nov-2017 16:00:00    'F.N'    'Market Price'      '-5'       'Intraday 1Hour'    12.32
```

Display three rows of IBM aggregated intraday data.

```
d(48:50,:)
```

```
ans =

  3×8 timetable

          Time            x_RIC        Domain        GMTOffset          Type           Open
    ____________________    _____    _____________    _________    _________________    _____

    06-Nov-2017 14:00:00    'IBM.N'    'Market Price'      '-5'       'Intraday 1Hour'    151.68
    06-Nov-2017 15:00:00    'IBM.N'    'Market Price'      '-5'       'Intraday 1Hour'    151.13
    06-Nov-2017 16:00:00    'IBM.N'    'Market Price'      '-5'       'Intraday 1Hour'    150.78
```

`d` is a timetable that contains these variables:

- Transaction date and time
- RIC
- Domain
- GMT time zone offset
- Aggregation type
- Open price
- High price
- Low price
- Last price

Input Arguments

c — Tick History from Refinitiv connection

connection object

Tick History from Refinitiv connection, specified as a connection object created with `trth`.

sec — Security

string array | cell array of character vectors

Security, specified as an `N`-by-2 string array or cell array of character vectors. The first column of the string array or cell array identifies the security. The second column identifies the type of security (for example, Reuters Instrument Code, or RIC).

Example: `["IBM.N","Ric"]`

Data Types: `string` | `cell`

`fields` — Fields
string array | cell array of character vectors

Fields, specified as a string array or cell array of character vectors. Specify Refinitiv intraday fields to retrieve intraday data. You can search for fields in MyRefinitiv.

Example: `["Trade - Exchange Time";"Trade - Price"]`

Data Types: `string` | `cell`

`startdate` — Start date
`datetime` array | string scalar | character vector | numeric scalar

Start date of the date range, specified as a `datetime` array, string scalar, character vector, or numeric scalar.

Example: `datetime('yesterday')`

Data Types: `double` | `char` | `string` | `datetime`

`enddate` — End date
`datetime` array | string scalar | character vector | numeric scalar

End date of the date range, specified as a `datetime` array, string scalar, character vector, or numeric scalar.

Example: `datetime('today')`

Data Types: `double` | `char` | `string` | `datetime`

`interval` — Aggregation interval
`'OneSecond'` | `'FiveSeconds'` | `'OneMinute'` | ...

Aggregation interval, specified as one of these values:

- `'OneSecond'`
- `'FiveSeconds'`
- `'OneMinute'`
- `'FiveMinutes'`
- `'TenMinutes'`
- `'FifteenMinutes'`
- `'OneHour'`

Specify these values as character vectors or string scalars.

Output Arguments

`d` — Intraday data
timetable

Intraday data, returned as a timetable with these variables:

- Transaction date and time
- RIC

- Domain
- GMT time zone offset
- Transaction type

Other variables in `d` include the specified fields in the `fields` input argument.

Version History

Introduced in R2018a

See Also

`trth` | `history`

Topics

“Decide to Buy Shares with Intraday Data Using Tick History from Refinitiv” on page 10-2

External Websites

Refinitiv REST API Documentation

quandl

Quandl connection

Description

The `quandl` function creates a `quandl` object, which represents a Quandl connection. To establish the connection, you must obtain a Quandl API key from the Quandl website by creating an account.

After you create a `quandl` object, you can use the `history` function to retrieve historical data.

Creation

Syntax

```
c = quandl(apikey)
```

Description

`c = quandl(apikey)` creates a Quandl connection using a Quandl API key.

Input Arguments

`apikey` — Quandl API key
character vector | string scalar

Quandl API key, specified as a character vector or string scalar. To obtain your API key, create an account using the Quandl website.

Data Types: `char` | `string`

Properties

`TimeOut` — Timeout
numeric scalar

Timeout, specified as a numeric scalar that indicates the number of seconds to wait for data to return before canceling the request.

Example: 15

Data Types: `double`

Object Functions

history Retrieve Quandl historical data

Examples

Retrieve Quandl Historical Data

Use a Quandl connection to retrieve historical data for a security within the available date range for the specified security.

Create a Quandl connection using a Quandl API key.

```
apikey = 'abcdef12345';
c = quandl(apikey)
```

```
c =

  quandl with properties:

    TimeOut: 100
```

`c` is the `quandl` connection object with the `TimeOut` property. The `TimeOut` property specifies waiting for a maximum of 100 seconds to return historical data before canceling the request.

Adjust the display format to display currency.

```
format bank
```

Retrieve historical data for the `CHRIS/ASX_WM2` security within the available date range for the security. This security provides historical future prices for Eastern Australian Wheat Futures, Continuous Contract #2. The specified security indicates the default periodicity (in this case, daily). `d` is a timetable with the time in the first variable and the previous settlement price in the second variable.

```
s = 'CHRIS/ASX_WM2';
d = history(c,s);
```

Display the first few rows of historical prices.

```
head(d)
```

```
ans =

  8×1 timetable

      Time        PreviousSettlement
    ___________    __________________

    28-Apr-2018          294.00
    27-Apr-2018          294.00
    26-Apr-2018          294.00
    25-Apr-2018          287.00
    24-Apr-2018          287.00
    23-Apr-2018          289.50
    21-Apr-2018          291.00
    20-Apr-2018          291.00
```

Decide to buy or sell this contract based on the historical prices.

Version History

Introduced in R2018b

See Also

Topics
“Access Quandl Error Messages” on page 11-2

External Websites
Quandl

history

Retrieve Quandl historical data

Syntax

```
d = history(c,s)
d = history(c,s,startdate,enddate)
d = history(c,s,startdate,enddate,[],
QueryName1,QueryValue1,...,QueryNameN,QueryValueN)
```

Description

`d = history(c,s)` retrieves Quandl historical data using a Quandl connection and a security.

`d = history(c,s,startdate,enddate)` also specifies a date range for the historical data to retrieve.

`d = history(c,s,startdate,enddate,[],
QueryName1,QueryValue1,...,QueryNameN,QueryValueN)` also specifies web service query parameters as one or more pairs of name-value arguments.

Examples

Retrieve Quandl Historical Data

Use a Quandl connection to retrieve historical data for a security within the available date range for the specified security.

Create a Quandl connection using a Quandl API key.

```
apikey = 'abcdef12345';
c = quandl(apikey)
```

```
c =

  quandl with properties:

    TimeOut: 100
```

`c` is the `quandl` connection object with the `TimeOut` property. The `TimeOut` property specifies waiting for a maximum of 100 seconds to return historical data before canceling the request.

Adjust the display format to display currency.

```
format bank
```

Retrieve historical data for the `CHRIS/ASX_WM2` security within the available date range for the security. This security provides historical future prices for Eastern Australian Wheat Futures, Continuous Contract #2. The specified security indicates the default periodicity (in this case, daily). `d`

is a timetable with the time in the first variable and the previous settlement price in the second variable.

```
s = 'CHRIS/ASX_WM2';
d = history(c,s);
```

Display the first few rows of historical prices.

```
head(d)
```

```
ans =

  8×1 timetable

      Time        PreviousSettlement
    ___________    __________________

    28-Apr-2018          294.00
    27-Apr-2018          294.00
    26-Apr-2018          294.00
    25-Apr-2018          287.00
    24-Apr-2018          287.00
    23-Apr-2018          289.50
    21-Apr-2018          291.00
    20-Apr-2018          291.00
```

Decide to buy or sell this contract based on the historical prices.

Retrieve Quandl Historical Data Within Date Range

Use a Quandl connection to retrieve historical data for a security within a specified date range.

Create a Quandl connection using a Quandl API key.

```
apikey = 'abcdef12345';
c = quandl(apikey);
```

Adjust the display format to display currency.

```
format bank
```

Retrieve historical data for the `CHRIS/ASX_WM2` security from March 1, 2018, through March 31, 2018. This security provides historical future prices for Eastern Australian Wheat Futures, Continuous Contract #2. The specified security indicates the default periodicity (in this case, daily). `d` is a timetable with the time in the first variable and the previous settlement price in the second variable.

```
s = 'CHRIS/ASX_WM2';
startdate = datetime('03-01-2018','InputFormat','MM-dd-yyyy');
enddate = datetime('03-31-2018','InputFormat','MM-dd-yyyy');
d = history(c,s,startdate,enddate);
```

Display the first few rows of historical prices.

```
head(d)
```

```
ans =

  8×1 timetable

      Time        PreviousSettlement
    ___________    __________________

    31-Mar-2018          277.50
    30-Mar-2018          277.50
    29-Mar-2018          277.50
    28-Mar-2018          278.50
    27-Mar-2018          278.00
    26-Mar-2018          278.50
    24-Mar-2018          280.00
    23-Mar-2018          280.00
```

Decide to buy or sell this contract based on the historical prices.

Input Arguments

c — Quandl connection

`quandl` object

Quandl connection, specified as a `quandl` object.

s — Security

character vector | string scalar

Security, specified as a character vector or string scalar.

Example: `"CHRIS/ASX_WM2"`

Data Types: `char` | `string`

startdate — Start date

`datetime` array | numeric scalar | string scalar | character vector

Start date of the date range, specified as a `datetime` array, numeric scalar, string scalar, or character vector. By default, the start date is the date of the first available historical data for the specified security `s`.

If you specify the `enddate` input argument, then you must specify the `startdate` input argument.

Example: `datetime('03-01-2018','InputFormat','MM-dd-yyyy')`

Example: `737121`

Data Types: `double` | `char` | `string` | `datetime`

enddate — End date

`datetime` array | numeric scalar | string scalar | character vector

End date of the date range, specified as a `datetime` array, numeric scalar, string scalar, or character vector. By default, the end date is the date of the last available historical data for the specified security `s`.

If you specify the `startdate` input argument, then you must specify the `enddate` input argument.

Example: `datetime('03-31-2018','InputFormat','MM-dd-yyyy')`

Example: `737181`

Data Types: `double` | `char` | `string` | `datetime`

`QueryName1,QueryValue1,...,QueryNameN,QueryValueN` — Web service query parameters
name-value pairs

Web service query parameters, specified as one or more pairs of name-value arguments. A `QueryName` argument is a character vector or string scalar that specifies the name of a query parameter. A `QueryValue` argument is a character vector or string scalar that specifies the value of the query parameter. Specify the name using a character vector or string scalar.

Example: `"TICKER","XYZ"` returns data for the XYZ ticker.

Data Types: `char` | `string`

Output Arguments

`d` — Quandl historical data
timetable | `matlab.net.http.ResponseMessage`

Quandl historical data, returned as a timetable or a `matlab.net.http.ResponseMessage` object.

If the historical data request is successful, then the `history` function returns data as a timetable. The timetable contains the time in the first variable. The subsequent variables in the timetable correspond to the returned data. The variables of the returned data depend on the specified security `s`.

If the historical data request is unsuccessful, the `history` function returns the error message in the `matlab.net.http.ResponseMessage` object. To access the error message, see “Access Quandl Error Messages” on page 11-2.

Version History

Introduced in R2018b

See Also

`quandl`

Topics
“Access Quandl Error Messages” on page 11-2

External Websites
Quandl

datastreamws

Datastream Web Services from Refinitiv connection

Description

The `datastreamws` function creates a `datastreamws` object, which represents a Datastream Web Services connection. You must obtain Datastream Web Services credentials. For credentials, contact Datastream Web Services. After you create a `datastreamws` object, you can retrieve historical data.

Creation

Syntax

```
c = datastreamws(username,password)
```

Description

`c = datastreamws(username,password)` creates a Datastream Web Services connection using a user name and password.

Input Arguments

`username` — User name
character vector | string scalar

User name, specified as a character vector or string scalar. For your user name, contact Datastream Web Services.

Example: `'ABCDEF'`

Data Types: `char` | `string`

`password` — Password
character vector | string scalar

Password, specified as a character vector or string scalar. For your password, contact Datastream Web Services.

Example: `'abcdef12345'`

Data Types: `char` | `string`

Properties

`Username` — User name
character vector

This property is read-only.

User name, specified as a character vector. For your user name, contact Datastream Web Services.

The `username` input argument sets the `Username` property.

Example: `'ABCDEF'`

Data Types: `char` | `string`

`TimeOut` — Timeout
100 (default) | numeric scalar

Timeout, specified as a numeric scalar that indicates the number of seconds to wait for data to return before canceling the request.

Example: `50`

Data Types: `double`

Object Functions

history Retrieve historical data from Datastream Web Services from Refinitiv

Examples

Retrieve Historical Data for Security

Use a Datastream Web Services connection to retrieve historical data for the specified security.

Create a Datastream Web Services connection using your user name and password.

```
username = 'ABCDEF';
password = 'abcdef12345';
c = datastreamws(username,password)
```

```
c =

  datastreamws with properties:

    Username: 'ABCDEF'
     TimeOut: 100
```

`c` is the `datastreamws` connection object with the `Username` and `TimeOut` properties. The `Username` property contains the specified user name. The `TimeOut` property specifies waiting for a maximum of 100 seconds to return historical data before canceling the request.

Adjust the display format to display currency.

```
format bank
```

Retrieve historical end-of-day price data for the last year. Specify the `VOD` security. `d` is a timetable that contains the date in the first variable and the end-of-day price in the second variable.

```
sec = 'VOD';
d = history(c,sec);
```

Display the first few prices.

```
head(d)
```

```
ans =

  8×1 timetable

           Time             VOD
    ____________________    ______

    03-May-2017 00:00:00    202.95
    04-May-2017 00:00:00    203.70
    05-May-2017 00:00:00    204.95
    08-May-2017 00:00:00    205.15
    09-May-2017 00:00:00    205.15
    10-May-2017 00:00:00    206.60
    11-May-2017 00:00:00    206.25
    12-May-2017 00:00:00    211.05
```

Use the end-of-day prices to make investment decisions for the `VOD` security.

Version History

Introduced in R2018b

See Also

Topics

“Retrieve Datastream Web Services Historical Data” on page 12-2
“Access Datastream Web Services Error Messages” on page 12-4

External Websites

Datastream Web Services from Refinitiv REST Service

history

Retrieve historical data from Datastream Web Services from Refinitiv

Syntax

```
d = history(c,sec)
d = history(c,sec,fields,date)
d = history(c,sec,fields,startdate,enddate)
d = history(c,sec,fields,startdate,enddate,period)
[d,response] = history( ___ )
```

Description

`d = history(c,sec)` retrieves historical Datastream Web Services data for a specified security for the last year.

`d = history(c,sec,fields,date)` retrieves historical data for specified fields and a specific date.

`d = history(c,sec,fields,startdate,enddate)` retrieves historical data for a date range.

`d = history(c,sec,fields,startdate,enddate,period)` retrieves historical data using a period.

`[d,response] = history( ___ )` returns a `ResponseMessage` object that contains an error message. To access the error message, see “Access Datastream Web Services Error Messages” on page 12-4.

Examples

Retrieve Historical Data for Security

Use a Datastream Web Services connection to retrieve historical data for the specified security.

Create a Datastream Web Services connection using your user name and password.

```
username = 'ABCDEF';
password = 'abcdef12345';
c = datastreamws(username,password)
```

```
c =

  datastreamws with properties:

    Username: 'ABCDEF'
     TimeOut: 100
```

`c` is the `datastreamws` connection object with the `Username` and `TimeOut` properties. The `Username` property contains the specified user name. The `TimeOut` property specifies waiting for a maximum of 100 seconds to return historical data before canceling the request.

Adjust the display format to display currency.

```
format bank
```

Retrieve historical end-of-day price data for the last year. Specify the `VOD` security. `d` is a timetable that contains the date in the first variable and the end-of-day price in the second variable.

```
sec = 'VOD';
d = history(c,sec);
```

Display the first few prices.

```
head(d)
```

```
ans =

  8×1 timetable

            Time              VOD
    ____________________    ______

    03-May-2017 00:00:00    202.95
    04-May-2017 00:00:00    203.70
    05-May-2017 00:00:00    204.95
    08-May-2017 00:00:00    205.15
    09-May-2017 00:00:00    205.15
    10-May-2017 00:00:00    206.60
    11-May-2017 00:00:00    206.25
    12-May-2017 00:00:00    211.05
```

Use the end-of-day prices to make investment decisions for the `VOD` security.

Return Historical Data for Specified Fields and Date

Use a Datastream Web Services connection to retrieve historical data for the specified security, fields, and date.

Create a Datastream Web Services connection using your user name and password. `c` is the `datastreamws` connection object.

```
username = 'ABCDEF';
password = 'abcdef12345';
c = datastreamws(username,password);
```

Adjust the display format to display currency.

```
format bank
```

Retrieve and display historical end-of-day price data for March 29, 2018. Specify the `VOD` security and these fields:

- Opening price
- High price
- Last closing price

`d` is a timetable with the date in the first variable and the fields in the subsequent variables.

```
sec = "VOD";
fields = ["PO";"PH";"P"];
date = datetime('03-29-2018','InputFormat','MM-dd-yyyy');
d = history(c,sec,fields,date)
```

```
d =

  1×3 timetable

            Time              PO        PH        P
    ____________________    ______    ______    ______

    29-Mar-2018 00:00:00    194.94    196.01    194.22
```

Use the end-of-day prices for each field to make investment decisions for the `VOD` security.

Return Historical Data for Date Range

Use a Datastream Web Services connection to retrieve historical data for the specified security, fields, and date range.

Create a Datastream Web Services connection using your user name and password. `c` is the `datastreamws` connection object.

```
username = 'ABCDEF';
password = 'abcdef12345';
c = datastreamws(username,password);
```

Adjust the display format to display currency.

```
format bank
```

Retrieve historical end-of-day price data from April 1, 2018, through April 30, 2018. Specify the `VOD` security and these fields:

- Opening price
- High price
- Last closing price

`d` is a timetable with the date in the first variable and the fields in the subsequent variables.

```
sec = "VOD";
fields = ["PO";"PH";"P"];
startdate = datetime('04-01-2018','InputFormat','MM-dd-yyyy');
enddate = datetime('04-30-2018','InputFormat','MM-dd-yyyy');
d = history(c,sec,fields,startdate,enddate);
```

Display the first few prices.

```
head(d)
```

```
ans =

  8×3 timetable

            Time              PO        PH        P
```

```
    ____________________    ______    ______    ______

    02-Apr-2018 00:00:00       NaN       NaN    194.22
    03-Apr-2018 00:00:00    193.70    194.15    193.90
    04-Apr-2018 00:00:00    196.64    198.10    197.22
    05-Apr-2018 00:00:00    200.45    203.90    203.65
    06-Apr-2018 00:00:00    203.15    205.15    204.00
    09-Apr-2018 00:00:00    204.35    205.45    203.65
    10-Apr-2018 00:00:00    204.45    205.90    205.60
    11-Apr-2018 00:00:00    205.50    207.70    206.30
```

Use the end-of-day prices for each field to make investment decisions for the `VOD` security.

Return Historical Data Using Period

Use a Datastream Web Services connection to retrieve historical data for the specified security, fields, date range, and period.

Create a Datastream Web Services connection using your user name and password. `c` is the `datastreamws` connection object.

```
username = 'ABCDEF';
password = 'abcdef12345';
c = datastreamws(username,password);
```

Adjust the display format to display currency.

```
format bank
```

Retrieve and display historical end-of-day price data from January 1, 2017, through December 31, 2017. Specify the `VOD` security and these fields:

- Opening price
- High price
- Last closing price

Specify a quarterly period. `d` is a timetable with the date in the first variable and the fields in the subsequent variables.

```
sec = "VOD";
fields = ["PO";"PH";"P"];
startdate = datetime('01-01-2017','InputFormat','MM-dd-yyyy');
enddate = datetime('12-31-2017','InputFormat','MM-dd-yyyy');
period = 'Q';
d = history(c,sec,fields,startdate,enddate,period)
```

```
d =

  4×3 timetable

           Time              PO        PH        P
    ____________________    ______    ______    ______

    01-Jan-2017 00:00:00       NaN       NaN    199.85
    01-Apr-2017 00:00:00    209.00    209.10    206.65
```

```
01-Jul-2017 00:00:00    217.65    219.20    218.70
01-Oct-2017 00:00:00    209.35    211.60    210.50
```

Use the quarterly prices for each field to make investment decisions for the `VOD` security.

Input Arguments

c — Datastream Web Services connection

`datastreamws` object

Datastream Web Services connection, specified as a `datastreamws` object.

sec — Security

character vector | cell array of character vectors | string scalar | string array

Security, specified as a character vector, cell array of character vectors, string scalar, or string array. Use a character vector or string scalar to specify one security. Use a cell array of character vectors or string array to specify multiple securities. For a constituent list, specify a single security in the `sec` input argument, for example, `"LFTSE100"`.

Example: `"VOD"`

Data Types: `char` | `string` | `cell`

fields — Field list

character vector | cell array of character vectors | string scalar | string array

Field list, specified as a character vector, cell array of character vectors, string scalar, or string array. Use a character vector or string scalar to specify one field. Use a cell array of character vectors or string array to specify multiple fields.

Example: `["PH","PO","P"]`

Data Types: `char` | `string` | `cell`

date — Date

`datetime` array | numeric scalar | string scalar | character vector

Date, specified as a `datetime` array, numeric scalar, string scalar, or character vector. Use this date to retrieve historical data for a specific day.

Example: `datetime('03-29-2018','InputFormat','MM-dd-yyyy')`

Data Types: `double` | `char` | `string` | `datetime`

startdate — Start date

`datetime` array | numeric scalar | string scalar | character vector

Start date of a date range, specified as a `datetime` array, numeric scalar, string scalar, or character vector. The default start date is the first date of available historical data for the specified security `sec`.

Example: `datetime('04-01-2018','InputFormat','MM-dd-yyyy')`

Data Types: `double` | `char` | `string` | `datetime`

enddate — End date

`datetime` array | numeric scalar | string scalar | character vector

End date of a date range, specified as a `datetime` array, numeric scalar, string scalar, or character vector. The default end date is the latest date of available historical data for the specified security `sec`.

Example: `datetime('04-30-2018','InputFormat','MM-dd-yyyy')`

Data Types: `double` | `char` | `string` | `datetime`

`period` — Period
`'D'` | `'W'` | `'M'` | `'Q'` | `'Y'`

Period, specified as one of these values:

- `'D'` — Daily
- `'W'` — Weekly
- `'M'` — Monthly
- `'Q'` — Quarterly
- `'Y'` — Yearly

You can specify the value as a character vector or string scalar. The default period depends on the specified security `sec`.

Output Arguments

`d` — Historical data
timetable | table

Historical data, returned as a timetable or table. The `history` function returns a timetable with data for one security. For multiple securities, the `history` function returns a timetable for the first syntax only and a table of nested timetables for the other syntaxes. To access one of the nested timetables, use dot notation, for example, `d.VOD`.

`response` — Response message
`matlab.net.http.ResponseMessage`

Response message, returned as a `matlab.net.http.ResponseMessage` object. The `ResponseMessage` object contains an error message. To access the error message, see “Access Datastream Web Services Error Messages” on page 12-4.

Version History

Introduced in R2018b

See Also

`datastreamws`

Topics

“Retrieve Datastream Web Services Historical Data” on page 12-2
“Access Datastream Web Services Error Messages” on page 12-4

External Websites
Datastream Web Services from Refinitiv REST Service

wind

WDS connection

Description

The `wind` function creates a `wind` object, which represents a Wind Data Feed Services (WDS) connection. First, open and log in to the Wind Financial Terminal, then create the `wind` object. You can use the object functions to retrieve current, historical, intraday, and real-time data from the Wind Financial Terminal. Also, you can create and delete orders and query order and account information in the Wind Financial Terminal. For details about WDS or the Wind Financial Terminal, see Wind Data Feed Services (WDS) .

Creation

Syntax

```
c = wind
```

Description

`c = wind` creates a WDS connection.

Object Functions

WDS Connection

close	Close WDS connection

WDS Data Retrieval

getdata	Current WDS data
history	Historical WDS data
timeseries	Intraday tick WDS data
realtime	Snapshot and subscription WDS data
stop	Cancel subscription WDS data request

WDS Order Management

createorder	Create WDS order
deleteorder	Cancel WDS order
query	Query WDS account and order information
tradelogin	Log in to WDS order management system
tradelogout	Log out from WDS order management system

Examples

Retrieve Current WDS Data for Security

Using a WDS connection, retrieve current data for a single security and display the data. Then close the connection.

Create a WDS connection.

```
c = wind;
```

Format output data for currency.

```
format bank
```

Using the `0001.HK` security, retrieve the current high and low prices.

```
s = '0001.HK';
f = ["high","low"];
d = getdata(c,s,f)
```

```
d=1×2 table
                HIGH      LOW
                _____    _____

    0001.HK    99.50    98.00
```

`d` is a table with one row for the single security. Each variable in the table corresponds to each specified field.

Close the WDS connection.

```
close(c)
```

Version History

Introduced in R2018a

See Also

Topics

“Decide to Buy Shares Using Current and Historical WDS Data” on page 14-2
“Create Order Using Real-Time Snapshot WDS Data” on page 14-4

External Websites

Wind Data Feed Services (WDS)

close

Close WDS connection

Syntax

```
close(c)
```

Description

`close(c)` closes the Wind Data Feed Services (WDS) connection.

Examples

Retrieve Current WDS Data for Security

Using a WDS connection, retrieve current data for a single security and display the data. Then close the connection.

Create a WDS connection.

```
c = wind;
```

Format output data for currency.

```
format bank
```

Using the `0001.HK` security, retrieve the current high and low prices.

```
s = '0001.HK';
f = ["high","low"];
d = getdata(c,s,f)
```

```
d=1×2 table
                HIGH      LOW
                _____    _____

    0001.HK    99.50    98.00
```

`d` is a table with one row for the single security. Each variable in the table corresponds to each specified field.

Close the WDS connection.

```
close(c)
```

Input Arguments

c — WDS connection

connection object

WDS connection, specified as a connection object created with the `wind` function.

Version History

Introduced in R2018a

See Also

`wind` | `getdata` | `history` | `timeseries` | `realtime` | `createorder`

Topics

“Decide to Buy Shares Using Current and Historical WDS Data” on page 14-2
“Create Order Using Real-Time Snapshot WDS Data” on page 14-4

External Websites

Wind Data Feed Services (WDS)

createorder

Create WDS order

Syntax

```
d = createorder(c,s,direction,price,quantity)
d = createorder(c,s,direction,price,quantity,Name,Value)
[d,e] = createorder( ___ )
```

Description

`d = createorder(c,s,direction,price,quantity)` returns order information after sending an order to the Wind Data Feed Services (WDS) order management system using the WDS connection. Specify the security, trade side, order price, and quantity of shares for the order.

`d = createorder(c,s,direction,price,quantity,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'TradePassword',"abcdefghi"` specifies the password for the WDS order management system.

`[d,e] = createorder( ___ )` also returns the WDS error identifier using any of the input argument combinations in the previous syntaxes. For troubleshooting, contact Wind Information Co., Ltd.

Examples

Create Order for Security

Using a WDS connection, log in to the order management system and create a buy order of a single security.

Create a WDS connection.

```
c = wind;
```

Log in to the WDS order management system using the WDS connection. Specify the broker, branch, user name, password, and account type.

```
broker = "0000";
branch = "0";
capitalaccount = "1234567891011";
password = "abcdefghi";
accttype = "SHSZ";
dlogin = tradelogin(c,broker,branch, ...
    capitalaccount,password,accttype);
```

Create a buy order of 100 shares of the `600000.SH` security using the WDS connection. Buy shares with the order price `12.0`, specified in the CNY currency.

```
s = '600000.SH';
direction = 'buy';
```

```
price = '12.0';
quantity = '100';
d = createorder(c,s,direction,price,quantity)
```

```
d =

  1×8 table

    RequestID    SecurityCode    TradeSide    OrderPrice    OrderVolume    LogonID    ErrorCode
    _________    ____________    _________    __________    ___________    _______    _________

       20        '600000.sh'       'BUY'        '12.0'         '100'         '1'          0
```

`d` is a table with these variables:

- Request identifier
- Security code
- Trade side
- Order price
- Order volume
- Login identifier
- Error code
- Error message

Query for the order status of the executed order and display the status. The order status `'Normal'` indicates a successful order execution.

```
d = query(c,'Order');
d.OrderStatus
```

```
d =

  'Normal'
```

This result assumes that the WDS order management system contains only one valid order execution.

Log out from the WDS order management system using the login identifier returned by the `tradelogin` function.

```
logonid = dlogin.LogonID;
d = tradelogout(c,logonid);
```

Close the WDS connection.

```
close(c)
```

Create Order for Security Using Credentials

Using a WDS connection, log in to the order management system and create a buy order of a single security. Use name-value pair arguments to specify the login identifier and password.

Create a WDS connection.

```
c = wind;
```

Log in to the WDS order management system using the WDS connection. Specify the broker, branch, user name, password, and account type.

```
broker = "0000";
branch = "0";
capitalaccount = "1234567891011";
password = "abcdefghi";
accttype = "SHSZ";
dlogin = tradelogin(c,broker,branch, ...
    capitalaccount,password,accttype);
```

Create a buy order of 100 shares of the `600000.SH` security using the WDS connection. Buy shares with the order price `12.0`, specified in the CNY currency. Use the `'LogonID'` and `'TradePassword'` name-value pair arguments to specify the login identifier and password.

```
s = '600000.SH';
direction = 'buy';
price = '12.0';
quantity = '100';
logonid = '1';
password = "abcdefghi";
d = createorder(c,s,direction,price,quantity, ...
    'LogonID',logonid,'TradePassword',password)
```

```
d =

  1×8 table

    RequestID    SecurityCode    TradeSide    OrderPrice    OrderVolume    LogonID    ErrorCode
    _________    ____________    _________    __________    ___________    _______    _________

       20        '600000.sh'       'BUY'        '12.0'         '100'         '1'           0
```

`d` is a table with these variables:

- Request identifier
- Security code
- Trade side
- Order price
- Order volume
- Login identifier
- Error code
- Error message

Query for the order status of the executed order and display the status. The order status `'Normal'` indicates a successful order execution.

```
d = query(c,'Order');
d.OrderStatus
```

This result assumes that the WDS order management system contains only one valid order execution.

```
d =

  'Normal'
```

Log out from the WDS order management system using the login identifier returned by the `tradelogin` function.

```
logonid = dlogin.LogonID;
d = tradelogout(c,logonid);
```

Close the WDS connection.

```
close(c)
```

Input Arguments

`c` — WDS connection

connection object

WDS connection, specified as a connection object created with the `wind` function.

`s` — Security

character vector | string scalar

Security, specified as a character vector or string scalar.

Example: `'0001.HK'`

Data Types: `char` | `string`

`direction` — Trade side

`'Buy'` | `'Short'` | `'Cover'` | ...

Trade side of the order, specified as one of these values:

- `'Buy'`
- `'BuyCollateral'`
- `'Cover'`
- `'CoverCovered'`
- `'CoverToday'`
- `'Merge'`
- `'Redemption'`
- `'Sell'`
- `'SellCollateral'`
- `'SellToday'`
- `'Short'`
- `'ShortCovered'`
- `'Split'`
- `'Subscription'`

The values for the `direction` input argument depend on the instrument type.

Instrument Type	Values
Stocks	`'Buy'` or `'Sell'` — Buy or sell stocks
Futures and options	• `'Buy'` — Buy long • `'Sell'` — Sell long • `'Short'` — Buy short • `'Cover'` — Sell short
`SHF` futures only	• `'Buy'` — Buy long • `'Sell'` — Sell long position yesterday or before • `'SellToday'` — Sell long position today • `'Short'` — Buy short • `'Cover'` — Sell short position yesterday or before • `'CoverToday'` — Sell short position today
`SHO` options only	• `'Buy'` — Buy long • `'Sell'` — Sell long • `'Short'` — Buy short • `'ShortCovered'` — Buy short with frozen underlying stock (not frozen margin) • `'Cover'` — Sell short • `'CoverCovered'` — Sell short covered
Short margin	• `'Buy'` — Margin purchase • `'Sell'` — Repayment • `'Short'` — Short sale • `'Cover'` — Return stock • `'BuyCollateral'` — Buy collateral • `'SellCollateral'` — Sell collateral
Funds and split-capital funds	• `'Buy'` — Buy fund in floor trading • `'Sell'` — Sell fund in floor trading • `'Subscription'` — Buy fund in OTC • `'Redemption'` — Sell fund in OTC
Split-capital funds only	• `'Merge'` — SCT merge to Fund of Funds • `'Split'` — Fund of Funds split to SCT

`price` — Order price

character vector | string scalar

Order price, specified as a character vector or string scalar. Specify the price of the order in the CNY currency.

Example: `'12.0'`

Data Types: `char` | `string`

quantity — Order quantity
character vector | string scalar

Order quantity, specified as a character vector or string scalar. Specify the number of shares for the order transaction.

Example: `'100'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `d = createorder(c,'600000.SH','buy','12.0','100','OrderType','LMT')` returns order information after sending a limit order of the `600000.SH` security to the WDS order management system. This order buys 100 shares of the security with an order price of 12, specified in CNY currency.

OrderType — Order type
`'LMT'` | `'B5TC'` | `'B5TL'`

Order type, specified as the comma-separated pair consisting of `'OrderType'` and one of these values.

Value	Description
`'LMT'`	Limit
`'BOC'`	Best of counterparty
`'BOP'`	Best of party
`'ITC'`	Immediately then cancel
`'B5TC'`	Best 5 then cancel
`'FOK'`	Fill or kill
`'B5TL'`	Best 5 then limit

For details about these values, contact Wind Information Co., Ltd.

HedgeType — Hedge type
`'SPEC'` | `'HEDG'`

Hedge type, specified as the comma-separated pair consisting of `'HedgeType'` and `'SPEC'` for speculation or `'HEDG'` for hedging (when trading futures).

For details about these values, contact Wind Information Co., Ltd.

LogonID — Login identifier
character vector | string scalar

Login identifier, specified as the comma-separated pair consisting of `'LogonID'` and a character vector or string scalar. Set the value of the `'LogonID'` name-value pair argument by using the `LogonID` variable in the `d` output argument of the `tradelogin` function.

Example: `'1'`

Data Types: `char` | `string`

`TradePassword` — Account password
character vector | string scalar

Account password, specified as the comma-separated pair consisting of `'TradePassword'` and a character vector or string scalar. For credentials, contact Wind Information Co., Ltd.

Example: `"abcdefghi"`

Data Types: `char` | `string`

`FundsType` — Fund type
`'ETF'`

Fund type, specified as the comma-separated pair consisting of `'FundsType'` and `'ETF'`.

For details about this value, contact Wind Information Co., Ltd.

`PortfolioNo` — Portfolio number
character vector | string scalar

Portfolio number, specified as the comma-separated pair consisting of `'PortfolioNo'` and a character vector or string scalar.

Example: `'3'`

Data Types: `char` | `string`

Output Arguments

`d` — Order information
table

Order information, returned as a table. The variables in the table depend on the specified order.

For details about the variables in the table, contact Wind Information Co., Ltd.

`e` — WDS error identifier
numeric scalar

WDS error identifier, returned as a numeric scalar. The value `0` indicates a successful execution of the `createorder` function. Otherwise, for details about the error, contact Wind Information Co., Ltd.

Version History

Introduced in R2018a

See Also

`wind` | `close` | `query` | `tradelogin` | `tradelogout`

Topics
“Create Order Using Real-Time Snapshot WDS Data” on page 14-4

External Websites
Wind Data Feed Services (WDS)

deleteorder

Cancel WDS order

Syntax

```
d = deleteorder(c,orderno)
d = deleteorder(c,orderno,Name,Value)
[d,e] = deleteorder( ___ )
```

Description

`d = deleteorder(c,orderno)` cancels a Wind Data Feed Services (WDS) order using the WDS connection.

`d = deleteorder(c,orderno,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'TradePassword',"abcdefghi"` specifies the password for the WDS order management system.

`[d,e] = deleteorder( ___ )` also returns the WDS error identifier using any of the input argument combinations in the previous syntaxes. For troubleshooting, contact Wind Information Co., Ltd.

Examples

Delete Order

Using a WDS connection, log in to the order management system, create a buy order for a single security, and delete the order.

Create a WDS connection.

```
c = wind;
```

Log in to the WDS order management system using the WDS connection. Specify the broker, branch, user name, password, and account type.

```
broker = "0000";
branch = "0";
capitalaccount = "1234567891011";
password = "abcdefghi";
accttype = "SHSZ";
dlogin = tradelogin(c,broker,branch, ...
    capitalaccount,password,accttype);
```

Create a buy order of 100 shares of the `600000.SH` security using the WDS connection. Buy shares with the order price `12.0`, specified in the CNY currency.

```
s = '600000.SH';
direction = 'buy';
price = '12.0';
```

```
quantity = '100';
d = createorder(c,s,direction,price,quantity);
```

Query for the order number of the executed order and display the number.

```
d = query(c,'Order');
orderno = d.OrderNumber
```

```
orderno =

  '12'
```

This result assumes that the WDS order management system contains only one valid order execution.

Delete the order using the WDS connection and the order number.

```
d = deleteorder(c,orderno)
```

```
d =

  1×3 table

    OrderNumber    ErrorCode      ErrorMsg
    ___________    _________    _____________

       '12'            0        'Sending ...'
```

`d` is a table that contains these variables:

- Order number
- Error code
- Error message

Log out from the WDS order management system using the login identifier returned by the `tradelogin` function.

```
logonid = dlogin.LogonID;
d = tradelogout(c,logonid);
```

Close the WDS connection.

```
close(c)
```

Delete Order Using Password

Using a WDS connection, log in to the order management system, create a buy order of a single security, and delete the order by using the account password.

Create a WDS connection.

```
c = wind;
```

Log in to the WDS order management system using the WDS connection. Specify the broker, branch, user name, password, and account type.

```
broker = "0000";
branch = "0";
```

```
capitalaccount = "1234567891011";
password = "abcdefghi";
accttype = "SHSZ";
dlogin = tradelogin(c,broker,branch, ...
    capitalaccount,password,accttype);
```

Create a buy order of 100 shares of the `600000.SH` security using the WDS connection. Buy shares with the order price `12.0`, specified in the CNY currency.

```
s = '600000.SH';
direction = 'buy';
price = '12.0';
quantity = '100';
d = createorder(c,s,direction,price,quantity);
```

Query for the order number of the executed order and display the number.

```
d = query(c,'Order');
orderno = d.OrderNumber
```

```
orderno =

  '12'
```

This result assumes that the WDS order management system contains only one valid order execution.

Delete the order using the WDS connection and the order number. Specify the account password using the `'TradePassword'` name-value pair argument.

```
d = deleteorder(c,orderno,'TradePassword',password)
```

```
d =

  1×3 table

    OrderNumber    ErrorCode      ErrorMsg
    ___________    _________    _____________

       '12'            0        'Sending ...'
```

`d` is a table that contains these variables:

- Order number
- Error code
- Error message

Log out from the WDS order management system using the login identifier returned by the `tradelogin` function.

```
logonid = dlogin.LogonID;
d = tradelogout(c,logonid);
```

Close the WDS connection.

```
close(c)
```

Input Arguments

`c` — WDS connection

connection object

WDS connection, specified as a connection object created with the `wind` function.

`orderno` — Order number

character vector | string scalar

Order number, specified as a character vector or string scalar. To find the order number, use the `query` function with the query term `'Order'`.

Example: `'12'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `d = deleteorder(c,'12','LogonID','1','TradePassword',"abcdefghi")` cancels the order, identified by the order number `'12'`, in the WDS order management system using the login identifier `'1'` and the account password `"abcdefghi"`.

`MarketType` — Security market identifier

`'SZ'` | `'SH'` | `'OC'` | ...

Security market identifier, specified as one of these values.

Value	Description
`'SZ'`	Shenzhen Stock Exchange
`'SH'`	Shanghai Stock Exchange
`'OC'`	National Equities Exchange and Quotations
`'HK'`	Hong Kong Stock Exchange
`'CZC'`	Zhengzhou Commodity Exchange
`'SHF'`	Shanghai Futures Exchange
`'DCE'`	Dalian Commodity Exchange
`'CFE'`	China Financial Futures Exchange

`LogonID` — Login identifier

character vector | string scalar

Login identifier, specified as the comma-separated pair consisting of `'LogonID'` and a character vector or string scalar. Set the value of the `'LogonID'` name-value pair argument by using the `LogonID` variable in the `d` output argument of the `tradelogin` function.

Example: `'1'`

Data Types: `char` | `string`

`TradePassword` — Account password
character vector | string scalar

Account password, specified as the comma-separated pair consisting of `'TradePassword'` and a character vector or string scalar. For credentials, contact Wind Information Co., Ltd.

Example: `"abcdefghi"`

Data Types: `char` | `string`

`OrderPrice` — Order price
character vector | string scalar

Order price, specified as a character vector or string scalar. Specify the order price in the CNY currency.

For HK only, use the `'OrderPrice'` name-value pair argument to change the price of an existing order. If the `'OrderPrice'` and `'OrderVolume'` name-value pair arguments are not specified, then the Wind Financial Terminal cancels the order.

Example: `'30'`

Data Types: `char` | `string`

`OrderVolume` — Order volume
character vector | string scalar

Order volume, specified as a character vector or string scalar.

For HK only, use the `'OrderVolume'` name-value pair argument to change the volume of an existing order. If the `'OrderPrice'` and `'OrderVolume'` name-value pair arguments are not specified, then the Wind Financial Terminal cancels the order.

Example: `'100'`

Data Types: `char` | `string`

Output Arguments

`d` — Deletion information
table

Deletion information, returned as a table with these variables:

- Order number
- Error code
- Error message

`e` — WDS error identifier
numeric scalar

WDS error identifier, returned as a numeric scalar. The value `0` indicates a successful execution of the `deleteorder` function. Otherwise, for details about the error, contact Wind Information Co., Ltd.

Version History

Introduced in R2018a

See Also

`wind` | `close` | `createorder` | `query` | `tradelogin` | `tradelogout`

Topics

“Create Order Using Real-Time Snapshot WDS Data” on page 14-4

External Websites

Wind Data Feed Services (WDS)

getdata

Current WDS data

Syntax

```
d = getdata(c,s,f)
d = getdata(c,s,f,Name,Value)
[d,e] = getdata( ___ )
```

Description

`d = getdata(c,s,f)` returns the current Wind Data Feed Services (WDS) market data for the specified securities and fields using the WDS connection.

`d = getdata(c,s,f,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'TradeDate',datetime('today')` returns market data for the current day.

`[d,e] = getdata( ___ )` also returns the WDS error identifier using any of the input argument combinations in the previous syntaxes. For troubleshooting, contact Wind Information Co., Ltd.

Examples

Retrieve Current WDS Data for Security

Using a WDS connection, retrieve current data for a single security and display the data. Then close the connection.

Create a WDS connection.

```
c = wind;
```

Format output data for currency.

```
format bank
```

Using the `0001.HK` security, retrieve the current high and low prices.

```
s = '0001.HK';
f = ["high","low"];
d = getdata(c,s,f)
```

```
d=1×2 table
                HIGH      LOW
                _____    _____

    0001.HK    99.50    98.00
```

`d` is a table with one row for the single security. Each variable in the table corresponds to each specified field.

Close the WDS connection.

```
close(c)
```

Retrieve Daily Current WDS Data

Using a WDS connection, retrieve current data for a single security for the day and display the data.

Create a WDS connection.

```
c = wind;
```

Format output data for currency.

```
format bank
```

For the `0001.HK` security, retrieve the high and low prices for the day using the WDS connection. Use the name-value pair argument `'Cycle'` to specify the period.

```
s = {'0001.HK'};
f = ["high","low"];
d = getdata(c,s,f,'Cycle','D')
```

```
d=1×2 table
                HIGH      LOW
                _____    _____

    0001.HK    99.50    98.00
```

`d` is a table with a row for the security. The variables in the table correspond to the specified fields.

Close the WDS connection.

```
close(c)
```

Input Arguments

c — WDS connection

connection object

WDS connection, specified as a connection object created with the `wind` function.

s — Securities

character vector | string scalar | cell array of character vectors | string array

Securities, specified as a character vector, string scalar, cell array of character vectors, or string array. For a single security, use a character vector or string scalar. For multiple securities, use a cell array of character vectors or string array.

Example: `'0001.HK'`

Data Types: `char` | `string` | `cell`

`f` — Fields

character vector | string scalar | cell array of character vectors | string array

Fields, specified as a character vector, string scalar, cell array of character vectors, or string array. For a single field, use a character vector or string scalar. For multiple fields, use a cell array of character vectors or string array.

For details about valid fields, contact Wind Information Co., Ltd.

Example: `{"high","low"}`

Data Types: `char` | `string` | `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `getdata(c,s,f,'TradeDate',datetime('yesterday'))` retrieves current WDS market data for yesterday.

`TradeDate` — Trade date

`datetime` scalar | numeric scalar | character vector | string scalar

Trade date, specified as the comma-separated pair consisting of `'TradeDate'` and a `datetime` scalar, numeric scalar, character vector, or string scalar.

If you do not specify a date, the `getdata` function sets the trade date to the current day.

Example: `731878`

Example: `datetime('yesterday')`

Data Types: `datetime` | `double` | `char` | `string`

`PriceAdj` — Price adjustment

`'N'` | `'F'` | `'B'` | `'T'`

Price adjustment, specified as the comma-separated pair consisting of `'PriceAdj'` and one of these values.

Value	Description
`'N'`	No
`'F'`	Forward
`'B'`	Backward
`'T'`	As per selected ex-rights time

For details about these values, contact Wind Information Co., Ltd.

`Cycle` — Cycle

`'D'` | `'W'` | `'M'` | ...

Cycle, specified as the comma-separated pair consisting of `'Cycle'` and one of these values.

Value	Description
'D'	Daily
'W'	Weekly
'M'	Monthly
'Q'	Quarterly
'S'	Semi-annually
'Y'	Annually

For details about these values, contact Wind Information Co., Ltd.

Output Arguments

d — Current WDS market data

table

Current WDS market data, returned as a table. The rows in the table correspond to the securities specified in the `s` input argument. The variables in the table correspond to the fields specified in the `f` input argument.

e — WDS error identifier

numeric scalar

WDS error identifier, returned as a numeric scalar. The value `0` indicates a successful execution of the `getdata` function. Otherwise, for details about the error, contact Wind Information Co., Ltd.

Version History

Introduced in R2018a

See Also

`wind` | `history` | `timeseries` | `realtime` | `createorder` | `close`

Topics

“Decide to Buy Shares Using Current and Historical WDS Data” on page 14-2

External Websites

Wind Data Feed Services (WDS)

history

Historical WDS data

Syntax

```
d = history(c,s,f,startdate,enddate)
d = history(c,s,f,startdate,enddate,Name,Value)
[d,e] = history( ___ )
```

Description

`d = history(c,s,f,startdate,enddate)` returns the historical Wind Data Feed Services (WDS) market data for the specified security and fields using the WDS connection. Specify a date range for the historical data to return.

`d = history(c,s,f,startdate,enddate,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'Currency','EUR'` returns data in the Euro currency.

`[d,e] = history( ___ )` also returns the WDS error identifier using any of the input argument combinations in the previous syntaxes. For troubleshooting, contact Wind Information Co., Ltd.

Examples

Retrieve Historical WDS Data for Security

Using a WDS connection, retrieve historical data for a single security and display the data.

Create a WDS connection.

```
c = wind;
```

Format output data for currency.

```
format bank
```

Using the `0001.HK` security, retrieve the open, high, low, and closing prices from August 10, 2017 through August 15, 2017.

```
s = '0001.HK';
f = ["open","high","low","close"];
startdate = '2017-08-10';
enddate = '2017-08-15';
d = history(c,s,f,startdate,enddate)
```

```
d=4×4 timetable
            Time              OPEN      HIGH      LOW      CLOSE
    ____________________    ______    ______    ______    ______

    10-Aug-2017 00:00:00    104.50    105.00    103.30    103.30
    11-Aug-2017 00:00:00    102.00    102.70    101.00    101.10
```

```
    14-Aug-2017 00:00:00    102.10    102.20    101.30    102.00
    15-Aug-2017 00:00:00    101.40    102.50    101.20    102.00
```

d is a timetable that contains one row for each trading day with the time and a variable for each specified field.

Close the WDS connection.

```
close(c)
```

Retrieve Historical WDS Data in Specified Currency

Using a WDS connection, retrieve historical data for a single security and display the data. Specify the currency for the data.

Create a WDS connection.

```
c = wind;
```

Format output data for currency.

```
format bank
```

Using the `0001.HK` security, retrieve the open, high, low, and closing prices from August 10, 2017 through August 15, 2017. Specify the `EUR` currency by using the `'Currency'` name-value pair argument.

```
s = '0001.HK';
f = ["open","high","low","close"];
startdate = '2017-08-10';
enddate = '2017-08-15';
currency = 'EUR';
d = history(c,s,f,startdate,enddate,'Currency',currency)
```

```
d=4×4 timetable
            Time             OPEN     HIGH      LOW     CLOSE
    _____________________    _____    _____    _____    _____

    10-Aug-2017 00:00:00    11.37    11.43    11.24    11.24
    11-Aug-2017 00:00:00    11.10    11.18    10.99    11.00
    14-Aug-2017 00:00:00    11.05    11.06    10.97    11.04
    15-Aug-2017 00:00:00    11.01    11.13    10.99    11.07
```

d is a timetable that contains one row for each trading day with the time and a variable for each specified field.

Close the WDS connection.

```
close(c)
```

Input Arguments

c — WDS connection

connection object

WDS connection, specified as a connection object created with the `wind` function.

s — Security

character vector | string scalar

Security, specified as a character vector or string scalar.

Example: `'0001.HK'`

Data Types: `char` | `string`

f — Fields

character vector | string scalar | cell array of character vectors | string array

Fields, specified as a character vector, string scalar, cell array of character vectors, or string array. For a single field, use a character vector or string scalar. For multiple fields, use a cell array of character vectors or string array.

For details about valid fields, contact Wind Information Co., Ltd.

Example: `{"high","low"}`

Data Types: `char` | `string` | `cell`

startdate — Start date

`datetime` scalar | numeric scalar | character vector | string scalar

Start date of the historical date range, specified as a `datetime` scalar, numeric scalar, character vector, or string scalar.

Example: `731878`

Example: `datetime('yesterday')`

Data Types: `datetime` | `double` | `char` | `string`

enddate — End date

`datetime` scalar | numeric scalar | character vector | string scalar

End date of the historical date range, specified as a `datetime` scalar, numeric scalar, character vector, or string scalar.

Example: `731878`

Example: `datetime('today')`

Data Types: `datetime` | `double` | `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `history(c,s,f,'Days','Weekdays','Currency','EUR')` returns historical WDS market data only for weekdays and in the Euro currency.

`Currency` — Currency
character vector | string scalar

Currency, specified as the comma-separated pair consisting of `'Currency'` and a character vector or string scalar that contains three characters identifying the ISO code for the currency. For example, specify `'USD'` for the US currency.

Data Types: `char` | `string`

`Days` — Days
`'Alldays'` (default) | `'Weekdays'`

Days, specified as the comma-separated pair consisting of `'Days'` and the value `'Alldays'` to return data for all days, or the value `'Weekdays'` to return data for weekdays only.

`Fill` — Fill
`'Null'` (default) | `'Previous'`

Fill, specified as the comma-separated pair consisting of `'Fill'` and the value `'Null'` to fill missing data with `NULL` values, or the value `'Previous'` to fill missing data with previous values.

`Period` — Period
`'D'` | `'W'` | `'M'` | ...

Period, specified as the comma-separated pair consisting of `'Period'` and one of these values.

Value	Description
`'D'`	Daily
`'W'`	Weekly
`'M'`	Monthly
`'Q'`	Quarterly
`'Y'`	Annually

For details about these values, contact Wind Information Co., Ltd.

`PriceAdj` — Price adjustment
`'F'` | `'B'` | `'T'` | ...

Price adjustment, specified as the comma-separated pair consisting of `'PriceAdj'` and one of these values.

Value	Description
`'F'`	Forward
`'B'`	Backward
`'T'`	Fixed
`'CP'`	Clean price
`'DP'`	Dirty price

Value	Description
`'MP'`	Market price
`'YTM'`	Yield

For details about these values, contact Wind Information Co., Ltd.

`TradingCalendar` — Exchange code
character vector | string scalar

Exchange code, specified as the comma-separated pair consisting of `'TradingCalendar'` and a character vector or string scalar. For example, specify `'NYSE'` for the New York Stock Exchange.

Data Types: `char` | `string`

Output Arguments

`d` — Historical WDS market data
timetable

Historical WDS market data, returned as a timetable. The rows in the timetable correspond to the dates in the date range, as specified by the `startdate` and `enddate` input arguments. The variables in the timetable correspond to the specified fields in the `f` input argument.

`e` — WDS error identifier
numeric scalar

WDS error identifier, returned as a numeric scalar. The value `0` indicates a successful execution of the `history` function. Otherwise, for details about the error, contact Wind Information Co., Ltd.

Version History

Introduced in R2018a

See Also

`wind` | `getdata` | `timeseries` | `realtime` | `createorder` | `close`

Topics
“Decide to Buy Shares Using Current and Historical WDS Data” on page 14-2

External Websites
Wind Data Feed Services (WDS)

query

Query WDS account and order information

Syntax

```
d = query(c,q)
d = query(c,q,Name,Value)
[d,e] = query(___)
```

Description

`d = query(c,q)` returns account, order, and portfolio information associated with a Wind Data Feed Services (WDS) account using the WDS connection and a query term.

`d = query(c,q,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'LogonID','1'` returns information filtered by the login identifier.

`[d,e] = query(___)` returns the WDS error identifier using any of the input argument combinations in the previous syntaxes. For troubleshooting, contact Wind Information Co., Ltd.

Examples

Query Account Information

Using a WDS connection, log in to the WDS order management system and query for account information.

Create a WDS connection.

```
c = wind;
```

Log in to the WDS order management system using the WDS connection. Specify the broker, branch, user name, password, and account type.

```
broker = "0000";
branch = "0";
capitalaccount = "1234567891011";
password = "abcdefghi";
accttype = "SHSZ";
dlogin = tradelogin(c,broker,branch, ...
    capitalaccount,password,accttype)
```

```
d =

  1×5 table

    LogonID     LogonAccount      AccountType    ErrorCode    ErrorMsg
    _______    _______________    ___________    _________    ________

       1       '1234567891011'      'SZSHA'          0           ''
```

`d` is a table with these variables:

- Login identifier
- Account number
- Account type
- Error code
- Error message

If the error code is `0` and the message is an empty character vector, then the login is successful.

Query for account information using the WDS connection and the `Account` query term.

```
q = 'Account';
d = query(c,q)
```

```
d =

  4×10 table

    ShareholderStatus    MainShareholderFlag    AccountType    MarketType    Shareholder      As
    _________________    ___________________    ___________    __________    ____________    ___

           48                    0                'SZSHA'        'SH'       '0123456789'    '12
           48                    0                'SHB'          'SH'       '0123456789'    '12
           48                    0                'SZSHA'        'SZ'       '0123456789'    '12
           48                    0                'SZB'          'SZ'       '0123456789'    '12
```

`d` is a table with these variables:

- Shareholder status
- Shareholder flag
- Account type
- Market type
- Shareholder
- Account number
- Customer number
- Seat
- Error code
- Error message

Log out from the WDS order management system using the login identifier returned by the `tradelogin` function.

```
logonid = dlogin.LogonID;
d = tradelogout(c,logonid)
```

```
d =

  1×3 table

    LogonID    ErrorCode    ErrorMsg
```

```
    ________    _________    ________

      '1'           0        'logout'
```

d is a table with these variables:

- Login identifier
- Error code
- Error message

Close the WDS connection.

```
close(c)
```

Query Account Information Using Login Identifier

Using a WDS connection, log in to the WDS order management system and query for account information by using the login identifier.

Create a WDS connection.

```
c = wind;
```

Log in to the WDS order management system using the WDS connection. Specify the broker, branch, user name, password, and account type.

```
broker = "0000";
branch = "0";
capitalaccount = "1234567891011";
password = "abcdefghi";
accttype = "SHSZ";
d = tradelogin(c,broker,branch, ...
    capitalaccount,password,accttype)
```

```
d =

  1×5 table

    LogonID     LogonAccount       AccountType    ErrorCode    ErrorMsg
    _______    _________________    ___________    _________    ________

       1       '1234567891011'       'SZSHA'          0           ''
```

d is a table with these variables:

- Login identifier
- Account number
- Account type
- Error code
- Error message

If the error code is 0 and the message is an empty character vector, then the login is successful.

Query for account information using the WDS connection, `Account` query term, and login identifier. Use the login identifier returned by the `tradelogin` function with the `'LogonID'` name-value pair argument.

```
q = 'Account';
logonid = d.LogonID;
d = query(c,q,'LogonID',logonid)
```

```
d =

  4×10 table

    ShareholderStatus    MainShareholderFlag    AccountType    MarketType    Shareholder      Ass
    _________________    ___________________    ___________    __________    ____________    ____

           48                     0               'SZSHA'         'SH'       '0123456789'    '123
           48                     0               'SHB'           'SH'       '0123456789'    '123
           48                     0               'SZSHA'         'SZ'       '0123456789'    '123
           48                     0               'SZB'           'SZ'       '0123456789'    '123
```

`d` is a table with these variables:

- Shareholder status
- Shareholder flag
- Account type
- Market type
- Shareholder
- Account number
- Customer number
- Seat
- Error code
- Error message

Log out from the WDS order management system using the login identifier.

```
d = tradelogout(c,logonid)
```

```
d =

  1×3 table

    LogonID    ErrorCode    ErrorMsg
    _______    _________    ________

      '1'          0        'logout'
```

`d` is a table with these variables:

- Login identifier
- Error code
- Error message

Close the WDS connection.

```
close(c)
```

Input Arguments

c — WDS connection

connection object

WDS connection, specified as a connection object created with the `wind` function.

q — Query term

`'Capital'` | `'Position'` | `'Order'` | ...

Query term, specified as one of these values.

Query Term Value	Description
`'Account'`	WDS account information
`'Capital'`	Current account values
`'CreditFund'`	Credit status associated with the WDS account
`'CreditPos'`	Credit position
`'Liabilities'`	Debt status associated with the WDS account
`'LogonID'`	User name information
`'Order'`	Orders associated with the WDS account
`'Portfolio'`	Portfolio information from asset management system
`'Position'`	Portfolio positions associated with the WDS account
`'ShortInfo'`	Securities lending information
`'Trade'`	Trading information for the current day

You can specify these values using character vectors or string scalars.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `d = query(c,'Order','LogonID','1','OrderNumber','12')` returns order information, filtered by the login identifier, for orders that have order number `'12'`.

LogonID — Login identifier

character vector | string scalar

Login identifier, specified as the comma-separated pair consisting of `'LogonID'` and a character vector or string scalar. Set the value of the `'LogonID'` name-value pair argument by using the `LogonID` variable in the `d` output argument of the `tradelogin` function.

For example, `d = query(c,'Order','LogonID','1')` returns order information only for the orders associated with the login identifier `'1'`.

Example: `'1'`

Data Types: `char` | `string`

`RequestID` — Request identifier
character vector | string scalar

Request identifier, specified as the comma-separated pair consisting of `'RequestID'` and a character vector or string scalar. Set the value of the `'RequestID'` name-value pair argument by using the `RequestID` variable in the `d` output argument of the `createorder` function.

For example, `d = query(c,'Order','RequestID','12')` returns order information only for the orders associated with the request identifier `'12'`.

Example: `"27"`

Data Types: `double`

`OrderNumber` — Order number
character vector | string scalar

Order number, specified as the comma-separated pair consisting of `'OrderNumber'` and a character vector or string scalar. To find the value, set the `q` input argument of the `query` function to `'Order'`. Then, use the `OrderNumber` variable in the `d` output argument of the `query` function.

For example, `d = query(c,'Order','OrderNumber','10')` returns order information only for the orders associated with the order number `'10'`.

Example: `"6"`

Data Types: `double`

`OrderType` — Order type
`'All'` (default) | `'Withdrawable'`

Order type, specified as the comma-separated pair consisting of `'OrderType'` and the value `'All'` for all orders or `'Withdrawable'` for orders that can be withdrawn.

For example, `d = query(c,'Order','OrderType','All')` returns all order types.

`PortfolioNo` — Portfolio number
character vector | string scalar

Portfolio number, specified as the comma-separated pair consisting of `'PortfolioNo'` and a character vector or string scalar.

For example, `d = query(c,'Portfolio','PortfolioNo','3')` returns portfolio information for the portfolio number `'3'`.

Example: `'3'`

Data Types: `char` | `string`

Output Arguments

`d` — Account information
table

Account information about the account, order, and portfolio, returned as a table. The variables in the table depend on the specified query in the `q` input argument.

For details about these variables, contact Wind Information Co., Ltd.

`e` — WDS error identifier
numeric scalar

WDS error identifier, returned as a numeric scalar. The value `0` indicates a successful execution of the `query` function. Otherwise, for details about the error, contact Wind Information Co., Ltd.

Version History

Introduced in R2018a

See Also

`wind` | `close` | `createorder` | `tradelogin` | `tradelogout`

Topics

“Create Order Using Real-Time Snapshot WDS Data” on page 14-4

External Websites

Wind Data Feed Services (WDS)

realtime

Snapshot and subscription WDS data

Syntax

```
d = realtime(c,s,f)
[d,e] = realtime(c,s,f)

requestid = realtime(c,s,f,eventhandler)
[requestid,e] = realtime(c,s,f,eventhandler)
```

Description

`d = realtime(c,s,f)` returns the real-time snapshot Wind Data Feed Services (WDS) data for the specified securities and fields using the WDS connection.

`[d,e] = realtime(c,s,f)` also returns the WDS error identifier. For troubleshooting, contact Wind Information Co., Ltd.

`requestid = realtime(c,s,f,eventhandler)` subscribes to the specified securities by using the specified fields and an event handler function.

`[requestid,e] = realtime(c,s,f,eventhandler)` also returns the WDS error identifier.

Examples

Retrieve WDS Snapshot Data

Using a WDS connection, retrieve snapshot data for two securities.

Create a WDS connection.

```
c = wind;
```

Format output data for currency.

```
format bank
```

Using the `0001.HK` and `0003.HK` securities and the WDS connection, retrieve real-time data for the last price and volume fields.

```
s = {'0001.HK','0003.HK'};
f = {'rt_last','rt_vol'};

d = realtime(c,s,f)
```

```
d =

  2×3 timetable

            Time              Codes       RT_LAST       RT_VOL
```

```
    ____________________    _________    _______    ___________

    28-Nov-2017 10:54:14    '0001.HK'     97.75      3199866.00
    28-Nov-2017 10:54:14    '0003.HK'     15.28     19995745.00
```

`d` is a timetable that contains rows for each security with the time and these variables:

- Security
- Last price
- Volume

Close the WDS connection.

```
close(c)
```

Retrieve WDS Subscription Data

Using a WDS connection, subscribe to two securities and process real-time events by using an event handler function. Then cancel the subscription.

Create a WDS connection.

```
c = wind;
```

Format output data for currency.

```
format bank
```

Using the `0002.HK` and `0003.HK` securities and the WDS connection, retrieve real-time data for the last price, volume, and last volume fields. Process real-time data events using the sample event handler function `windEventHandler`. You can use the sample event handler function or create a custom event handler function to process events.

```
s = {'0002.HK','0003.HK'};
f = {'rt_last','rt_vol','rt_last_vol'}};

requestid = realtime(c,s,f,@(varargin)windEventHandler(varargin))
```

```
requestid =

  uint64

   5
```

`requestid` is the request identifier associated with the subscription. The event handler function `windEventHandler` creates a variable in the MATLAB workspace named `winddata`. This variable contains the subscription data.

Display the subscription data.

```
winddata
```

```
winddata =

  2×4 timetable
```

```
        Time             Codes       RT_LAST      RT_VOL       RT_LAST_VOL
    _____________________    _________    _______    ___________    ___________

    28-Nov-2017 10:55:25    '0002.HK'     81.30      2106274.00     422500.00
    28-Nov-2017 10:55:25    '0003.HK'     15.28     19995745.00    1398000.00
```

`winddata` is a timetable that contains a row for each security with the time and these variables:

- Security
- Last price
- Volume
- Last volume

Stop the data subscription using the request identifier.

```
stop(c,requestid)
```

Close the WDS connection.

```
close(c)
```

Input Arguments

`c` — WDS connection

connection object

WDS connection, specified as a connection object created with the `wind` function.

`s` — Securities

character vector | string scalar | cell array of character vectors | string array

Securities, specified as a character vector, string scalar, cell array of character vectors, or string array. For a single security, use a character vector or string scalar. For multiple securities, use a cell array of character vectors or string array.

Example: `'0001.HK'`

Data Types: `char` | `string` | `cell`

`f` — Fields

character vector | string scalar | cell array of character vectors | string array

Fields, specified as a character vector, string scalar, cell array of character vectors, or string array. For a single field, use a character vector or string scalar. For multiple fields, use a cell array of character vectors or string array.

For details about valid fields, contact Wind Information Co., Ltd.

Example: `{"high","low"}`

Data Types: `char` | `string` | `cell`

`eventhandler` — Event handler function

function handle

Event handler function, specified as a function handle. You can use the example event handling function `windEventHandler` to process real-time WDS events. Or, you can define a custom event handler function to process events of your choice.

The event handler function `windEventHandler` creates the variable `winddata` in the MATLAB workspace. The `windEventHandler` function returns `winddata` as a timetable that contains real-time WDS data. If an error occurs, the function returns `winddata` as a table that contains an error code. For troubleshooting, contact Wind Information Co., Ltd.

The `winddata` timetable contains rows for each real-time WDS event with the time. The first variable in this timetable is the specified securities in the `s` input argument. The remaining variables are the specified fields in the `f` input argument.

To access the code of the function, enter `edit windEventHandler` at the command line.

To define a custom event handler function:

1. Choose the WDS events to process, monitor, or evaluate.
2. Decide how the custom event handler processes these events.
3. Determine the input and output arguments for the custom event handler function.
4. Write the code for the custom event handler function. For details, see “Create Functions in Files”.

After defining the function, you can run it by passing the name of the function as a function handle. For details about function handles, see “Create Function Handle”.

Example: `@(varargin)windEventHandler(varargin)`

Data Types: `function_handle`

Output Arguments

d — Real-time snapshot WDS data

timetable

Real-time snapshot WDS data, returned as a timetable. The rows of the timetable correspond to the real-time snapshots with the time. The first variable in the timetable is the specified securities in the `s` input argument. The remaining variables in the timetable are the specified fields in the `f` input argument.

requestid — Request identifier

numeric scalar

Request identifier for the real-time data subscription, returned as a numeric scalar. To stop the real-time data subscription, specify the `requestid` output argument in the `stop` function.

e — WDS error identifier

numeric scalar

WDS error identifier, returned as a numeric scalar. The value `0` indicates a successful execution of the `realtime` function. Otherwise, for details about the error, contact Wind Information Co., Ltd.

Version History

Introduced in R2018a

See Also

`wind` | `getdata` | `history` | `timeseries` | `createorder` | `close` | `stop`

Topics

“Create Order Using Real-Time Snapshot WDS Data” on page 14-4

External Websites

Wind Data Feed Services (WDS)

stop

Cancel subscription WDS data request

Syntax

```
stop(c,requestid)
```

Description

`stop(c,requestid)` cancels the Wind Data Feed Services (WDS) subscription data request specified by the request identifier using the WDS connection.

Examples

Retrieve WDS Subscription Data

Using a WDS connection, subscribe to two securities and process real-time events by using an event handler function. Then cancel the subscription.

Create a WDS connection.

```
c = wind;
```

Format output data for currency.

```
format bank
```

Using the `0002.HK` and `0003.HK` securities and the WDS connection, retrieve real-time data for the last price, volume, and last volume fields. Process real-time data events using the sample event handler function `windEventHandler`. You can use the sample event handler function or create a custom event handler function to process events.

```
s = {'0002.HK','0003.HK'};
f = {'rt_last','rt_vol','rt_last_vol'}};

requestid = realtime(c,s,f,@(varargin)windEventHandler(varargin))
```

```
requestid =

  uint64

   5
```

`requestid` is the request identifier associated with the subscription. The event handler function `windEventHandler` creates a variable in the MATLAB workspace named `winddata`. This variable contains the subscription data.

Display the subscription data.

```
winddata
```

```
winddata =

  2×4 timetable

          Time              Codes       RT_LAST      RT_VOL       RT_LAST_VOL
    ____________________    _________    _______    ___________    ___________

    28-Nov-2017 10:55:25    '0002.HK'     81.30      2106274.00     422500.00
    28-Nov-2017 10:55:25    '0003.HK'     15.28     19995745.00    1398000.00
```

`winddata` is a timetable that contains a row for each security with the time and these variables:

- Security
- Last price
- Volume
- Last volume

Stop the data subscription using the request identifier.

```
stop(c,requestid)
```

Close the WDS connection.

```
close(c)
```

Input Arguments

`c` — WDS connection

connection object

WDS connection, specified as a connection object created with the `wind` function.

`requestid` — Request identifier

numeric scalar

Request identifier, specified as a numeric scalar created by the `realtime` function.

Example: 5

Data Types: `uint64`

Version History

Introduced in R2018a

See Also

`wind` | `realtime` | `close`

Topics

“Create Order Using Real-Time Snapshot WDS Data” on page 14-4

External Websites

Wind Data Feed Services (WDS)

timeseries

Intraday tick WDS data

Syntax

```
d = timeseries(c,s,f,t)
d = timeseries(c,s,f,{startdate,enddate})
d = timeseries(c,s,f,{startdate,enddate},interval)
d = timeseries(c,s,f,{startdate,enddate},interval,Name,Value)
[d,e] = timeseries( ___ )
```

Description

`d = timeseries(c,s,f,t)` returns raw intraday tick Wind Data Feed Services (WDS) data for the specified security, fields, and date using the WDS connection.

`d = timeseries(c,s,f,{startdate,enddate})` returns raw WDS intraday tick data for the specified date range.

`d = timeseries(c,s,f,{startdate,enddate},interval)` specifies an interval for the intraday data to return.

`d = timeseries(c,s,f,{startdate,enddate},interval,Name,Value)` specifies additional options using one or more name-value pair arguments. These options specify a time range for each day in the specified date range. For example, `'PeriodStart',datetime('10:30:00')` sets a time range that starts at 10:30 AM and ends at the end of the trading day.

`[d,e] = timeseries( ___ )` also returns the WDS error identifier using any of the input argument combinations in the previous syntaxes. For troubleshooting, contact Wind Information Co., Ltd.

Examples

Retrieve Intraday Tick WDS Data

Using a WDS connection, retrieve intraday tick data for a single security and display the data.

Create a WDS connection.

```
c = wind;
```

Format output data for currency.

```
format bank
```

For the `600000.SH` security, retrieve the intraday tick data for high and low prices. Retrieve ticks for the current day using the WDS connection.

```
s = {'600000.SH'};
f = ["high","low"];
```

```
t = datetime('now');
d = timeseries(c,s,f,t);
```

d is a timetable that contains a row for each tick with the time and a variable for each specified field.

Display the first three rows of intraday tick data.

```
head(d,3)
```

```
ans=3×2 timetable
            Time             high      low
    _____________________    _____    _____

    28-Nov-2017 13:17:42    13.07    12.92
    28-Nov-2017 13:17:45    13.07    12.92
    28-Nov-2017 13:17:48    13.07    12.92
```

Close the WDS connection.

```
close(c)
```

Retrieve Intraday Tick WDS Data Using Date Range

Using a WDS connection, retrieve intraday tick data for a single security and display the data. Specify a date range for the intraday tick data to return.

Create a WDS connection.

```
c = wind;
```

Format output data for currency.

```
format bank
```

For the 600000.SH security, retrieve the intraday tick data for high and low prices. Retrieve ticks from November 20, 2017 through November 23, 2017 using the WDS connection.

```
s = {'600000.SH'};
f = ["high","low"];
startdate = datetime('2017-11-20');
enddate = datetime('2017-11-23');
d = timeseries(c,s,f,{startdate,enddate});
```

d is a timetable that contains a row for each tick with the time and a variable for each specified field.

Display the last eight rows of intraday tick data.

```
tail(d)
```

```
ans=8×2 timetable
            Time             high      low
    _____________________    _____    _____

    22-Nov-2017 14:59:46    13.44    13.00
    22-Nov-2017 14:59:49    13.44    13.00
```

```
    22-Nov-2017 14:59:52    13.44    13.00
    22-Nov-2017 14:59:55    13.44    13.00
    22-Nov-2017 14:59:58    13.44    13.00
    22-Nov-2017 15:00:01    13.44    13.00
    22-Nov-2017 15:00:02    13.44    13.00
    22-Nov-2017 15:00:02    13.44    13.00
```

Close the WDS connection.

```
close(c)
```

Retrieve Intraday Tick WDS Data Using Interval

Using a WDS connection, retrieve intraday tick data for a single security and display the data. Specify a date range for the intraday tick data to return. Also, specify the interval to aggregate the tick data.

Create a WDS connection.

```
c = wind;
```

Format output data for currency.

```
format bank
```

For the `600000.SH` security, retrieve the intraday tick data for high and low prices. Retrieve ticks from November 20, 2017 through November 23, 2017 using the WDS connection. Specify 1-minute bars to aggregate the data.

```
s = {'600000.SH'};
f = ["high","low"];
startdate = datetime('2017-11-20');
enddate = datetime('2017-11-23');
interval = 1;
d = timeseries(c,s,f,{startdate,enddate},interval);
```

`d` is a timetable that contains a row for each aggregated tick with the time and a variable for each specified field.

Display the last eight rows of the aggregated intraday tick data.

```
tail(d)
```

```
ans=8×2 timetable
            Time             high      low
    _____________________    _____    _____

    22-Nov-2017 14:53:00    13.22    13.21
    22-Nov-2017 14:54:00    13.23    13.21
    22-Nov-2017 14:55:00    13.23    13.22
    22-Nov-2017 14:56:00    13.23    13.22
    22-Nov-2017 14:57:00    13.23    13.22
    22-Nov-2017 14:58:00    13.23    13.22
    22-Nov-2017 14:59:00    13.24    13.21
    22-Nov-2017 15:00:00    13.23    13.23
```

Close the WDS connection.

```
close(c)
```

Retrieve Intraday Tick WDS Data Using Time Range

Using a WDS connection, retrieve intraday tick data for a single security and display the data. Specify a date range for the intraday tick data to return. Also, specify the interval to aggregate the tick data. Then, specify the time range for each day in the date range.

Create a WDS connection.

```
c = wind;
```

Format output data for currency.

```
format bank
```

For the `600000.SH` security, retrieve the intraday tick data for high and low prices. Retrieve ticks from November 20, 2017 through November 23, 2017 using the WDS connection. Specify 1-minute bars to aggregate the data. Also, specify the time range from 9:30 AM through 10:30 AM using the `'PeriodStart'` and `'PeriodEnd'` name-value pair arguments.

```
s = {'600000.SH'};
f = ["high","low"];
startdate = datetime('2017-11-20');
enddate = datetime('2017-11-23');
interval = 1;
starttime = datetime('09:30:00');
endtime = datetime('10:30:00');
d = timeseries(c,s,f,{startdate,enddate},interval,'PeriodStart',starttime,'PeriodEnd',endtime);
```

`d` is a timetable that contains a row for each aggregated tick with the time and a variable for each specified field.

Display the first three rows of the aggregated intraday tick data.

```
head(d,3)
```

```
ans=3×2 timetable
            Time             high      low
    ____________________    _____    _____

    20-Nov-2017 09:30:00    12.72    12.68
    20-Nov-2017 09:31:00    12.75    12.71
    20-Nov-2017 09:32:00    12.77    12.73
```

Close the WDS connection.

```
close(c)
```

Input Arguments

`c` — WDS connection
connection object

WDS connection, specified as a connection object created with the `wind` function.

`s` — Security
character vector | string scalar

Security, specified as a character vector or string scalar.

Example: `'0001.HK'`

Data Types: `char` | `string`

`f` — Fields
character vector | string scalar | cell array of character vectors | string array

Fields, specified as a character vector, string scalar, cell array of character vectors, or string array. For a single field, use a character vector or string scalar. For multiple fields, use a cell array of character vectors or string array.

For details about valid fields, contact Wind Information Co., Ltd.

Example: `{"high","low"}`

Data Types: `char` | `string` | `cell`

`t` — Date
`datetime` scalar | numeric scalar | character vector | string scalar

Date, specified as a `datetime` scalar, numeric scalar, character vector, or string scalar.

Example: `datetime('today')`

Data Types: `datetime` | `double` | `char` | `string`

`startdate` — Start date
`datetime` scalar | numeric scalar | character vector | string scalar

Start date, specified as a `datetime` scalar, numeric scalar, character vector, or string scalar.

Example: `datetime('2017-08-10')`

Data Types: `datetime` | `double` | `char` | `string`

`enddate` — End date
`datetime` scalar | numeric scalar | character vector | string scalar

End date, specified as a `datetime` scalar, numeric scalar, character vector, or string scalar.

Example: `datetime('2017-08-19')`

Data Types: `datetime` | `double` | `char` | `string`

`interval` — Interval
numeric scalar

Interval for aggregating interval tick data into minute bars, specified as a numeric scalar.

Example: 1

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: d = timeseries(c,'0001.HK','open', {'2017-08-10','2017-08-19'},1,'PeriodStart',datetime('now')-.25,'PeriodEnd',datetime('now')) returns aggregated ticks for the open price in 1-minute bars for the 0001.HK security from August 10, 2017, through August 19, 2017. This syntax returns data for ticks that occur within 6 hours of the current time in each day.

PeriodStart — Start time
datetime scalar | numeric scalar | character vector | string scalar

Start time, specified as the comma-separated pair consisting of 'PeriodStart' and a datetime scalar, numeric scalar, character vector, or string scalar.

Use the 'PeriodStart' name-value pair argument with the PeriodEnd name-value pair argument to specify the time range for each day in the date range from startdate through enddate.

If you do not specify the 'PeriodEnd' name-value pair argument, the timeseries function uses the end of the trading day as the end of the time range.

Example: datetime('now')-.5

Data Types: datetime | double | char | string

PeriodEnd — End time
datetime scalar | numeric scalar | character vector | string scalar

End time, specified as the comma-separated pair consisting of 'PeriodEnd' and a datetime scalar, numeric scalar, character vector, or string scalar.

Use the 'PeriodEnd' name-value pair argument with the PeriodStart name-value pair argument to specify the time range for each day in the date range from startdate through enddate.

If you do not specify the 'PeriodStart' name-value pair argument, the timeseries function uses the start of the trading day as the start of the time range.

Example: datetime('now')

Data Types: datetime | double | char | string

Output Arguments

d — Intraday tick WDS data
timetable

Intraday tick WDS data, returned as a timetable. The rows of the timetable correspond to the date range specified by startdate and enddate and, optionally, the time range specified by the

`PeriodStart` and `PeriodEnd` name-value pair arguments. The variables of the timetable correspond to the fields specified in the `f` input argument.

e — WDS error identifier
numeric scalar

WDS error identifier, returned as a numeric scalar. The value `0` indicates a successful execution of the `timeseries` function. Otherwise, for details about the error, contact Wind Information Co., Ltd.

Version History

Introduced in R2018a

See Also

`wind` | `getdata` | `history` | `realtime` | `createorder` | `close`

Topics

“Decide to Buy Shares Using Current and Historical WDS Data” on page 14-2

External Websites

Wind Data Feed Services (WDS)

tradelogin

Log in to WDS order management system

Syntax

```
d = tradelogin(c,broker,branch,capitalaccount,password,accttype)
[d,e] = tradelogin(c,broker,branch,capitalaccount,password,accttype)
```

Description

`d = tradelogin(c,broker,branch,capitalaccount,password,accttype)` returns login information after logging in to the Wind Data Feed Services (WDS) order management system using:

- WDS connection
- Broker
- Branch
- Capital account
- Password
- Account type

`[d,e] = tradelogin(c,broker,branch,capitalaccount,password,accttype)` also returns the WDS error identifier. For troubleshooting, contact Wind Information Co., Ltd.

Examples

Query Account Information Using Login Identifier

Using a WDS connection, log in to the WDS order management system and query for account information by using the login identifier.

Create a WDS connection.

```
c = wind;
```

Log in to the WDS order management system using the WDS connection. Specify the broker, branch, user name, password, and account type.

```
broker = "0000";
branch = "0";
capitalaccount = "1234567891011";
password = "abcdefghi";
accttype = "SHSZ";
d = tradelogin(c,broker,branch, ...
    capitalaccount,password,accttype)
```

```
d =

  1×5 table
```

LogonID	LogonAccount	AccountType	ErrorCode	ErrorMsg
1	'1234567891011'	'SZSHA'	0	''

d is a table with these variables:

- Login identifier
- Account number
- Account type
- Error code
- Error message

If the error code is `0` and the message is an empty character vector, then the login is successful.

Query for account information using the WDS connection, `Account` query term, and login identifier. Use the login identifier returned by the `tradelogin` function with the `'LogonID'` name-value pair argument.

```
q = 'Account';
logonid = d.LogonID;
d = query(c,q,'LogonID',logonid)
```

```
d =

  4×10 table

    ShareholderStatus    MainShareholderFlag    AccountType    MarketType    Shareholder      Ass
    _________________    ___________________    ___________    __________    ___________    ____

           48                     0              'SZSHA'         'SH'       '0123456789'    '123
           48                     0              'SHB'           'SH'       '0123456789'    '123
           48                     0              'SZSHA'         'SZ'       '0123456789'    '123
           48                     0              'SZB'           'SZ'       '0123456789'    '123
```

d is a table with these variables:

- Shareholder status
- Shareholder flag
- Account type
- Market type
- Shareholder
- Account number
- Customer number
- Seat
- Error code
- Error message

Log out from the WDS order management system using the login identifier.

```
d = tradelogout(c,logonid)
```

```
d =

  1×3 table

    LogonID    ErrorCode    ErrorMsg
    _______    _________    ________

      '1'          0        'logout'
```

`d` is a table with these variables:

- Login identifier
- Error code
- Error message

Close the WDS connection.

```
close(c)
```

Input Arguments

`c` — WDS connection

connection object

WDS connection, specified as a connection object created with the `wind` function.

`broker` — Broker specification

character vector | string scalar

Broker specification, specified as a character vector or string scalar.

Example: `"0000"`

Data Types: `char` | `string`

`branch` — Branch name

character vector | string scalar

Branch name, specified as a character vector or string scalar.

Example: `"0"`

Data Types: `char` | `string`

`capitalaccount` — User name

character vector | string scalar

User name of the WDS account, specified as a character vector or string scalar. For credentials, contact Wind Information Co., Ltd.

Example: `"1234567891011"`

Data Types: `char` | `string`

`password` — Password

character vector | string scalar

Password of the WDS account, specified as a character vector or string scalar. For credentials, contact Wind Information Co., Ltd.

Example: `"abcdefghi"`

Data Types: `char` | `string`

`accttype` — Account type
character vector | string scalar

Account type, specified as a character vector or string scalar.

Example: `"SHSZ"`

Data Types: `char` | `string`

Output Arguments

`d` — Login information
table

Login information, returned as a table with these variables:

- Login identifier
- Account number
- Account type
- Error code
- Error message

`e` — WDS error identifier
numeric scalar

WDS error identifier, returned as a numeric scalar. The value `0` indicates a successful execution of the `tradelogin` function. Otherwise, for details about the error, contact Wind Information Co., Ltd.

Version History

Introduced in R2018a

See Also

`wind` | `close` | `createorder` | `query` | `tradelogout`

Topics

“Create Order Using Real-Time Snapshot WDS Data” on page 14-4

External Websites

Wind Data Feed Services (WDS)

tradelogout

Log out from WDS order management system

Syntax

```
d = tradelogout(c,logonid)
[d,e] = tradelogout(c,logonid)
```

Description

`d = tradelogout(c,logonid)` returns logout information after logging out from the Wind Data Feed Services (WDS) order management system using the WDS connection and the login identifier.

`[d,e] = tradelogout(c,logonid)` also returns the WDS error identifier. For troubleshooting, contact Wind Information Co., Ltd.

Examples

Query Account Information Using Login Identifier

Using a WDS connection, log in to the WDS order management system and query for account information by using the login identifier.

Create a WDS connection.

```
c = wind;
```

Log in to the WDS order management system using the WDS connection. Specify the broker, branch, user name, password, and account type.

```
broker = "0000";
branch = "0";
capitalaccount = "1234567891011";
password = "abcdefghi";
accttype = "SHSZ";
d = tradelogin(c,broker,branch, ...
    capitalaccount,password,accttype)
```

```
d =

  1×5 table

    LogonID     LogonAccount      AccountType    ErrorCode    ErrorMsg
    _______    _______________    ___________    _________    ________

       1       '1234567891011'      'SZSHA'          0           ''
```

`d` is a table with these variables:

- Login identifier

- Account number
- Account type
- Error code
- Error message

If the error code is `0` and the message is an empty character vector, then the login is successful.

Query for account information using the WDS connection, `Account` query term, and login identifier. Use the login identifier returned by the `tradelogin` function with the `'LogonID'` name-value pair argument.

```
q = 'Account';
logonid = d.LogonID;
d = query(c,q,'LogonID',logonid)
```

```
d =

  4×10 table

    ShareholderStatus    MainShareholderFlag    AccountType    MarketType    Shareholder      As
    _________________    ___________________    ___________    __________    ____________    ___

           48                    0                'SZSHA'         'SH'       '0123456789'    '12
           48                    0                'SHB'           'SH'       '0123456789'    '12
           48                    0                'SZSHA'         'SZ'       '0123456789'    '12
           48                    0                'SZB'           'SZ'       '0123456789'    '12
```

`d` is a table with these variables:

- Shareholder status
- Shareholder flag
- Account type
- Market type
- Shareholder
- Account number
- Customer number
- Seat
- Error code
- Error message

Log out from the WDS order management system using the login identifier.

```
d = tradelogout(c,logonid)
```

```
d =

  1×3 table

    LogonID    ErrorCode    ErrorMsg
    _______    _________    ________

      '1'          0        'logout'
```

`d` is a table with these variables:

- Login identifier
- Error code
- Error message

Close the WDS connection.

```
close(c)
```

Input Arguments

`c` — WDS connection

connection object

WDS connection, specified as a connection object created with the `wind` function.

`logonid` — Login identifier

character vector | string scalar

Login identifier, specified as a character vector or string scalar. Specify the value of the login identifier by using the `LogonID` variable in the `d` output argument of the `tradelogin` function. For example, enter `logonid = d.LogonID;` at the command line.

Example: `'1'`

Data Types: `char` | `string`

Output Arguments

`d` — Logout information

table

Logout information, returned as a table with these variables:

- Login identifier
- Error code
- Error message

`e` — WDS error identifier

numeric scalar

WDS error identifier, returned as a numeric scalar. The value `0` indicates a successful execution of the `tradelogout` function. Otherwise, for details about the error, contact Wind Information Co., Ltd.

Version History

Introduced in R2018a

See Also

`wind` | `close` | `createorder` | `query` | `tradelogin`

Topics
“Create Order Using Real-Time Snapshot WDS Data” on page 14-4

External Websites
Wind Data Feed Services (WDS)

xtrdr

Create X_TRADER connection

Description

The `xtrdr` function creates an `xtrdr` object, which represents an X_TRADER connection. After you create an `xtrdr` object, you can use the object functions to create instrument notifiers, instruments, order sets, and order profiles, and obtain current data. You can also submit orders to X_TRADER.

Note Create only one X_TRADER connection per MATLAB session. To create an X_TRADER connection, start a new MATLAB session.

Creation

Syntax

```
c = xtrdr
```

Description

`c = xtrdr` creates an X_TRADER connection object `c`. The `xtrdr` function starts X_TRADER or connects to an existing X_TRADER session.

Properties

`Gate` — Gate
ActiveX COM object

Gate, specified as an ActiveX COM object.

Example: `[1x1 COM.Xtapi_TTGate_1]`

`InstrNotify` — Instrument notifier
X_TRADER XTAPI instrument notifier object

Instrument notifier, specified as an X_TRADER XTAPI instrument notifier object. For details, see X_TRADER API.

To set this property, use the `createNotifier` function.

Example: `[1×1 COM.Xtapi_TTInstrNotify]`

`Instrument` — Instrument
X_TRADER XTAPI instrument object

Instrument, specified as an X_TRADER XTAPI instrument object. For details, see X_TRADER API.

To set this property, use the `createInstrument` function.

Example: `[1×1 COM.Xtapi_TTInstrObj]`

`OrderSet` — Order set
X_TRADER XTAPI order set object

Order set, specified as an X_TRADER order set object. For details, see X_TRADER API.

To set this property, use the `createOrderSet` function.

Example: `[1×1 COM.Xtapi_TTOrderSet]`

Object Functions

createNotifier	Create instrument notifier for X_TRADER
createInstrument	Create instrument for X_TRADER
createOrderSet	Create order set for X_TRADER
createOrderProfile	Create order profile for X_TRADER
getData	Obtain current X_TRADER data
close	Close X_TRADER connection

Examples

Create X_TRADER Connection

Use an X_TRADER connection to retrieve the exchange and last price data for an instrument. The instrument used in this example continually expires.

To ensure that you use a current instrument, see the **Market Explorer** in X_TRADER Pro.

Create an X_TRADER connection.

```
c = xtrdr
```

```
c =

  xtrdr with properties:

          Gate: [1x1 COM.Xtapi_TTGate_1]
    InstrNotify: []
     Instrument: []
       OrderSet: []
```

Define an input structure `s` with fields corresponding to valid X_TRADER API options. This example defines the input structure for Euro-Bobl Futures.

```
s = [];
s.Exchange = 'Eurex';
s.Product = 'OGBM';
s.ProdType = 'Option';
s.Contract = 'Jan12 P12300';
s.Alias = 'TestInstrument3';

s
```

```
s =
```

```
    Exchange: 'Eurex'
     Product: 'OGBM'
    ProdType: 'Option'
    Contract: 'Jan12 P12300'
       Alias: 'TestInstrument3'
```

Requirement: Restart the MATLAB session before reusing an `'Alias'` setting.

Create an X_TRADER instrument.

```
createInstrument(c,s)
```

Return the exchange and last price fields for the instrument.

```
s = c.Instrument(1);
f = {'Exchange','Last'};
d = getData(c,s,f)
```

```
d =

    Exchange: {'Eurex'}
        Last: {'45'}
```

Close the X_TRADER connection.

```
close(c)
```

Version History

Introduced in R2013a

See Also

Topics

“Workflows for Trading Technologies X_TRADER” on page 1-16
“Create Order Using X_TRADER” on page 1-13
“Listen for X_TRADER Price Updates” on page 1-18
“Listen for X_TRADER Price Market Depth Updates” on page 1-20
“Submit X_TRADER Orders” on page 1-23

External Websites

X_TRADER API

close

Close X_TRADER connection

Syntax

```
close(X)
```

Description

`close(X)` closes the X_TRADER connection `X`.

Examples

Close X_TRADER Connection

```
close(X)
```

Input Arguments

X — X_TRADER connection

connection object

X_TRADER connection, specified as a connection object created using `xtrdr`.

Version History

Introduced in R2013a

See Also

`xtrdr`

Topics

“Create Order Using X_TRADER” on page 1-13
“Listen for X_TRADER Price Updates” on page 1-18
“Listen for X_TRADER Price Market Depth Updates” on page 1-20
“Submit X_TRADER Orders” on page 1-23
“Workflows for Trading Technologies X_TRADER” on page 1-16

External Websites

X_TRADER API

createInstrument

Create instrument for X_TRADER

Syntax

```
createInstrument(c,s)
createInstrument(c,Name,Value)
```

Description

`createInstrument(c,s)` creates the X_TRADER instrument defined by the structure `s` with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

`createInstrument(c,Name,Value)` creates the instrument using one or more `Name,Value` pair arguments with names and values corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Examples

Create an X_TRADER Instrument Using an Input Structure

The instruments used in these examples continually expire. To ensure you use a current instrument, see the **Market Explorer** in X_TRADER Pro.

Create the X_TRADER connection.

```
c = xtrdr;
```

Define an input structure `s` with fields corresponding to valid X_TRADER API options. For example, create the input structure for Euro-Bobl Futures.

```
s = [];
s.Exchange = 'Eurex';
s.Product = 'OGBM';
s.ProdType = 'Option';
s.Contract = 'Jan12 P12300';
s.Alias = 'TestInstrument3';
s
```

```
s =

    Exchange: 'Eurex'
     Product: 'OGBM'
    ProdType: 'Option'
    Contract: 'Jan12 P12300'
       Alias: 'TestInstrument3'
```

Requirement: Restart the MATLAB session before reusing an `'Alias'` setting.

Create an X_TRADER instrument.

```
createInstrument(c,s)
```

Close the connection.

```
close(c)
```

Create an X_TRADER Instrument Using Name-Value Pairs

Create the X_TRADER connection.

```
c = xtrdr;
```

Create an X_TRADER instrument for Euro-Bobl Futures using name-value pair arguments corresponding to valid X_TRADER API options.

```
createInstrument(c,'Exchange','Eurex','Product','OGBM',...
                'ProdType','Option','Contract','Jan12 P12300',...
                'Alias','TestInstrument3')
```

Close the connection.

```
close(c)
```

Retrieve Data Using Multiple X_TRADER Instruments

Create the X_TRADER connection.

```
c = xtrdr;
```

Create an X_TRADER instrument for Euro-Bobl Futures using name-value pair arguments corresponding to valid X_TRADER API options.

```
createInstrument(c,'Exchange','Eurex','Product','OGBM',...
                'ProdType','Option','Contract','Jun14 P127',...
                'Alias','PriceInstrumentEurex')
```

Create another X_TRADER instrument for CAISO NP15 EZ Gen Hub 5 MW Peak Calendar-Day Real-Time LMP Futures using name-value pair arguments corresponding to valid X_TRADER API options. This contract expires in April 2014.

```
createInstrument(c,'Exchange','CME','Product','2F',...
                'ProdType','Future','Contract','Apr14',...
                'Alias','PriceInstrumentCMEApr14')
```

Create another X_TRADER instrument for CAISO NP15 EZ Gen Hub 5 MW Peak Calendar-Day Real-Time LMP Futures using name-value pair arguments corresponding to valid X_TRADER API options. This contract expires in October 2014.

```
createInstrument(c,'Exchange','CME','Product','2F',...
                'ProdType','Future','Contract','Oct14',...
                'Alias','PriceInstrumentCMEOct14')
```

Retrieve the exchange and product identifier for all three X_TRADER instruments.

```
d = getData(c,{'Exchange','Product'})

d =
    Exchange: {3x1 cell}
     Product: {3x1 cell}
```

`d` is a structure containing the `Exchange` and `Product` fields. The fields are cell arrays.

Display the `Exchange` field.

```
d.Exchange

ans =
    'Eurex'
    'CME'
    'CME'
```

The `Exchange` field contains the exchange names Eurex and CME for the three X_TRADER instruments.

Close the connection.

```
close(c)
```

Input Arguments

c — X_TRADER connection

connection object

X_TRADER connection, specified as a connection object created using `xtrdr`.

s — X_TRADER input structure

structure

X_TRADER input structure, specified using fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Caution If the symbols for the exchange are entered incorrectly or the exchange server is down, an error appears. For example, if the exchange is “CME” and the CME exchange server is down, then this error appears: The price server for the Exchange CME is down. Unable to create instrument.

Example: `s = [];`

```
s.Exchange = 'Eurex';
s.Product = 'OGBM';
s.ProdType = 'Option';
s.Contract = 'Jan12 P12300';
s.Alias = 'TestInstrument3';
```

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
```
createInstrument(X,'Exchange','Eurex','Product','OGBM','ProdType','Option','Contract','Jan12 P12300','Alias','TestInstrument3')
```

`Property1` — Valid X_TRADER API options
character vector | string scalar

Valid X_TRADER API options, specified as a character vector or string scalar using information in the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Requirements:

- When using the `'Alias'` name-value pair argument, ensure that every `'Alias'` name is unique across all X_TRADER instruments.
- Restart the MATLAB session before reusing an `'Alias'` name.

Otherwise, `createInstrument` returns an error.

Data Types: `char` | `string`

`Property2` — Valid X_TRADER API options
character vector | string scalar

Valid X_TRADER API options, specified as a character vector or string scalar using information in the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Data Types: `char` | `string`

Version History
Introduced in R2013a

See Also
`xtrdr` | `createNotifier` | `createOrderProfile` | `createOrderSet`

Topics
“Create Order Using X_TRADER” on page 1-13
“Listen for X_TRADER Price Updates” on page 1-18
“Listen for X_TRADER Price Market Depth Updates” on page 1-20
“Submit X_TRADER Orders” on page 1-23
“Workflows for Trading Technologies X_TRADER” on page 1-16

External Websites
X_TRADER API

createNotifier

Create instrument notifier for X_TRADER

Syntax

```
createNotifier(X,S)
createNotifier(X,Name,Value)
```

Description

`createNotifier(X,S)` creates the `xtrdr` instrument notifier defined by the structure `S` with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

`createNotifier(X,Name,Value)` creates the instrument notifier using X_TRADER API options specified by one or more `Name,Value` pair arguments with names and values corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Examples

Create an X_TRADER Instrument Notifier Using an Input Structure

Start X_TRADER.

```
X = xtrdr;
```

Define an input structure, `S`, with fields corresponding to valid X_TRADER API options.

```
S = [];
S.UpdateFilter = '';
S.EnablePriceUpdates = -1;
S.EnableDepthUpdates = 0;
S.DebugLogLevel = 3;
S.EnableOrderSetUpdates = -1;
S.DeliverAllPriceUpdates = 0;
S
```

```
S =

  struct with fields:

             UpdateFilter: ''
       EnablePriceUpdates: -1
       EnableDepthUpdates: 0
            DebugLogLevel: 3
    EnableOrderSetUpdates: -1
   DeliverAllPriceUpdates: 0
```

Create an `xtrdr` instrument notifier.

```
createNotifier(X,S)
```

Close the connection.

```
close(X)
```

Create an X_TRADER Instrument Notifier Using Name-Value Pairs

Start X_TRADER.

```
X = xtrdr;
```

Create an `xtrdr` instrument using name-value pairs corresponding to valid X_TRADER API options.

```
createNotifier(X,'UpdateFilter','','EnablePriceUpdates',-1, ...
    'EnableDepthUpdates',0,'DebugLogLevel',3, ...
    'EnableOrderSetUpdates',-1,'DeliverAllPriceUpdates',0)
```

Close the connection.

```
close(X)
```

Input Arguments

X — X_TRADER connection

connection object

X_TRADER connection, specified as a connection object created using `xtrdr`.

S — `xtrdr` input structure with fields

structure

`xtrdr` input structure, specified with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Example:
```
S = [];
S.Exchange = 'Eurex';
S.Product = 'OGBM';
S.ProdType = 'Option';
S.Contract = 'Jan12 P12300';
S.Alias = 'TestInstrument3';
```

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
```
createNotifier(X,'UpdateFilter','','EnablePriceUpdates',-1,'EnableDepthUpdate
```

`s',0,'DebugLogLevel',3,'EnableOrderSetUpdates',-1,'DeliverAllPriceUpdates',0)` creates the `xtrdr` instrument notifier using valid API options.

`Property1` — Valid X_TRADER API options
character vector | string scalar

Valid X_TRADER API options, specified as a character vector or string scalar using the details described in Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Example:
`createNotifier(X,'UpdateFilter','','EnablePriceUpdates',-1,'EnableDepthUpdates',0,'DebugLogLevel',3,'EnableOrderSetUpdates',-1,'DeliverAllPriceUpdates',0)`

Data Types: `char` | `string`

`Property2` — Valid X_TRADER API options
character vector | string scalar

Valid X_TRADER API options, specified as a character vector or string scalar using the details described in Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Example:
`createNotifier(X,'UpdateFilter','','EnablePriceUpdates',-1,'EnableDepthUpdates',0,'DebugLogLevel',3,'EnableOrderSetUpdates',-1,'DeliverAllPriceUpdates',0)`

Data Types: `char` | `string`

Version History
Introduced in R2013a

See Also
`xtrdr` | `createInstrument` | `createOrderProfile` | `createOrderSet`

Topics
“Listen for X_TRADER Price Updates” on page 1-18
“Listen for X_TRADER Price Market Depth Updates” on page 1-20
“Submit X_TRADER Orders” on page 1-23
“Workflows for Trading Technologies X_TRADER” on page 1-16

External Websites
X_TRADER API

createOrderProfile

Create order profile for X_TRADER

Syntax

```
P = createOrderProfile(X,S)
P = createOrderProfile(X,Name,Value)
```

Description

`P = createOrderProfile(X,S)` creates an order profile defined by the structure `S` with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

`P = createOrderProfile(X,Name,Value)` creates an order profile using X_TRADER API options specified by one or more `Name,Value` pair arguments with names and values corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Examples

Create an Order Profile Using an Input Structure

Start X_TRADER.

```
X = xtrdr;
```

Define an input structure, `S`, with fields corresponding to valid X_TRADER API options.

```
S = [];
S.Instrument = [];
S.Customer = '';
S.Alias = '';
S.ReadProperties = 'b';
S.WriteProperties = 'b';
S.Customers = {'<Default>'};
S.RoundOption = 2;
S.CustomerDefaults = [];
S
```

```
S =

          Instrument: []
            Customer: ''
               Alias: ''
      ReadProperties: 'b'
     WriteProperties: 'b'
           Customers: {'<Default>'}
         RoundOption: 2
    CustomerDefaults: []
```

Create an order profile.

```
P = createOrderProfile(X,S);
```

Close the connection.

```
close(X)
```

Create an Order Profile Using Name-Value Pairs

Start X_TRADER.

```
X = xtrdr;
```

Create an order profile using name-value pairs corresponding to valid X_TRADER API options.

```
createOrderProfile(X,'Instrument',[],'Customer','',...
             'Alias','','ReadProperties','b',...
             'WriteProperties','b','Customers',{'<Default>'},...
             'RoundOption',2,'CustomerDefaults',[])
```

Close the connection.

```
close(X)
```

Input Arguments

X — X_TRADER connection

connection object

X_TRADER connection, specified as a connection object created using `xtrdr`.

S — `xtrdr` input structure with fields

structure

`xtrdr` input structure, specified with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Example:
```
S = [];
S.Exchange = 'Eurex';
S.Product = 'OGBM';
S.ProdType = 'Option';
S.Contract = 'Jan12 P12300';
S.Alias = 'TestInstrument3';
```

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `createOrderProfile(X,'Instrument',[],'Customer','<Default>','Alias','','RoundOption',2,'CustomerDefaults')`

`Property1` — Valid X_TRADER API options
character vector | string scalar

Valid X_TRADER API options, specified as a character vector or string scalar using the details described in Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Example: `createOrderProfile(X,'Instrument',[],'Customer','<Default>','Alias','','RoundOption',2,'CustomerDefaults')`

Data Types: `char` | `string`

`Property2` — Valid X_TRADER API options
character vector | string scalar

Valid X_TRADER API options, specified as a character vector or string scalar using the details described in Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Example: `createOrderProfile(X,'Instrument',[],'Customer','<Default>','Alias','','RoundOption',2,'CustomerDefaults')`

Data Types: `char` | `string`

Output Arguments

`P` — Order profile
structure

Order profile, returned as a structure.

Version History

Introduced in R2013a

See Also

`xtrdr` | `createInstrument` | `createNotifier` | `createOrderSet`

Topics
“Create Order Using X_TRADER” on page 1-13
“Listen for X_TRADER Price Updates” on page 1-18
“Listen for X_TRADER Price Market Depth Updates” on page 1-20
“Submit X_TRADER Orders” on page 1-23
“Workflows for Trading Technologies X_TRADER” on page 1-16

External Websites
X_TRADER API

createOrderSet

Create order set for X_TRADER

Syntax

```
createOrderSet(X)
createOrderSet(X,S)
createOrderSet(X,Name,Value)
```

Description

`createOrderSet(X)` creates an `xtrdr` order set with empty properties. You can set the properties individually using X_TRADER API options. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

`createOrderSet(X,S)` creates an `xtrdr` order set defined by the structure `S` with fields corresponding to X_TRADER API options. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

`createOrderSet(X,Name,Value)` creates an order set using X_TRADER API options specified by one or more `Name,Value` pair arguments with names and values corresponding to X_TRADER API options. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Examples

Create an Empty Order Set

Start X_TRADER.

```
X = xtrdr;
```

Create an order set without any properties.

```
createOrderSet(X)
```

Close the connection.

```
close(X)
```

Create an Order Set Using an Input Structure

Start X_TRADER.

```
X = xtrdr;
```

Define an input structure, `S`, with fields corresponding to X_TRADER API options.

```
S = [];
S.Count = 0;
```

```
S.Alias = '';
S.ReadProperties = 'b';
S.WriteProperties = 'b';
S.EnableOrderSetUpdates = -1;
S.EnableOrderFillData = 0;
S.EnableOrderSend = 0;
S.EnableOrderAutoDelete = 0;
S.QuotingOrderProfile = [];
S.DebugLogLevel = 3;
S.QuoteWithCancelReplace = 0;
S.EnableOrderUpdateData = 0;
S.EnableFillCaching = 0;
S.AvgOpenPriceMode = 'NONE';
S.EnableOrderRejectData = 0;
S.OrderStatusNotifyMode = 'ORD_NOTIFY_NONE';
```

Create an order set.

```
createOrderSet(X,S)
```

Close the connection.

```
close(X)
```

Create an Order Set Using Name-Value Pair Arguments

Start X_TRADER.

```
X = xtrdr;
```

Create an order set using name-value pair arguments corresponding to X_TRADER API options.

```
createOrderSet(X,'Count',0,'Alias','','ReadProperties','b',...
              'WriteProperties','b','EnableOrderSetUpdates',-1,...
              'EnableOrderFillData',0,'EnableOrderSend',0,...
              'EnableOrderAutoDelete',0,'QuotingOrderProfile',[],...
              'DebugLogLevel,3,'QuoteWithCancelReplace',0,...
              'EnableOrderUpdateData',0,'EnableFillCaching',0,...
              'AvgOpenPriceMode','NONE','EnableOrderRejectData',0,...
              'OrderStatusNotifyMode','ORD_NOTIFY_NONE')
```

Close the connection.

```
close(X)
```

Input Arguments

X — X_TRADER connection

connection object

X_TRADER connection, specified as a connection object created using `xtrdr`.

S — X_TRADER API properties

structure

X_TRADER API properties, specified as a structure where the field names match the X_TRADER API properties. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Example: `S = [];`
```
S.Exchange = 'Eurex';
S.Product = 'OGBM';
S.ProdType = 'Option';
S.Contract = 'Jan12 P12300';
S.Alias = 'TestInstrument3';
```

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
```
createOrderSet(X,'Count',0,'Alias','','ReadProperties','b','WriteProperties',
'b','EnableOrderSetUpdates',-1,'EnableOrderFillData',0,'EnableOrderSend',0,'E
nableOrderAutoDelete',0,'QuotingOrderProfile',
[]'DebugLogLevel,3,'QuoteWithCancelReplace',0,'EnableOrderUpdateData',0,'Enab
leFillCaching',0,'AvgOpenPriceMode','NONE','EnableOrderRejectData',0,'OrderSt
atusNotifyMode','ORD_NOTIFY_NONE')
```

`Property1` — X_TRADER API options

character vector | string scalar

X_TRADER API options, specified as a character vector or string scalar using the details described in Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Data Types: `char` | `string`

`Property2` — X_TRADER API options

character vector | string scalar

X_TRADER API options, specified as a character vector or string scalar using the details described in Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Data Types: `char` | `string`

Version History

Introduced in R2013a

See Also

`xtrdr` | `createInstrument` | `createNotifier` | `createOrderProfile`

Topics

“Create Order Using X_TRADER” on page 1-13
“Listen for X_TRADER Price Updates” on page 1-18
“Listen for X_TRADER Price Market Depth Updates” on page 1-20
“Submit X_TRADER Orders” on page 1-23
“Workflows for Trading Technologies X_TRADER” on page 1-16

External Websites
X_TRADER API

getData

Obtain current X_TRADER data

Syntax

```
D = getData(X,S,F)
D = getData(X,F)
```

Description

`D = getData(X,S,F)` returns data for the fields `F` for the `xtrdr` instrument object, `S`, with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

`D = getData(X,F)` returns data for the fields `F` for all instruments associated with the `xtrdr` session object, `X`.

Examples

Return Exchange and Last Price for an Instrument

Return the exchange and last price fields for the instrument defined in `x.Instrument(1)`.

```
D = getData(X,X.Instrument(1),{'Exchange','Last'})
```

```
D =

    Exchange: {'CME'}
        Last: {'45'}
```

Return Exchange and Last Price for an Alias

Return the exchange and last price fields for the instrument defined by the alias `PriceInstrument1`.

```
D = getData(X,'PriceInstrument1',{'Exchange','Last'})
```

```
D =

    Exchange: {'CME'}
        Last: {'45'}
```

Return Exchange and Last Price for All Session Instruments

Return the exchange and last price fields for all instruments associated with the `xtrdr` session object, `X`.

```
D = getData(X,{'Exchange','Last'})

D =

    Exchange: {2x1 cell}
        Last: {2x1 cell}
```

Input Arguments

X — X_TRADER connection

connection object

X_TRADER connection, specified as a connection object created using `xtrdr`.

S — X_TRADER instrument

instrument object

X_TRADER instrument, specified as an instrument object created using `createInstrument` or aliases with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API RTD Tutorial* or *X_TRADER API Class Reference*.

Example: `x.Instrument(1)`

F — Fields for the instrument object

character vector | cell array of character vectors | string scalar | string array

Fields for the instrument object or aliases, `S`, specified as a character vector, cell array of character vectors, string scalar, or string array. `F` without a corresponding `S` are fields for all instruments associated with the `xtrdr` session object, `X`.

Example: `{'Exchange','Last'}`

Data Types: `char` | `cell` | `string`

Output Arguments

D — X_TRADER data

structure

X_TRADER data, returned as a structure. For missing data, `D` contains a `NaN`.

Version History

Introduced in R2013a

See Also

`xtrdr` | `createInstrument`

Topics

“Listen for X_TRADER Price Updates” on page 1-18
“Listen for X_TRADER Price Market Depth Updates” on page 1-20
“Submit X_TRADER Orders” on page 1-23
“Workflows for Trading Technologies X_TRADER” on page 1-16

External Websites
X_TRADER API