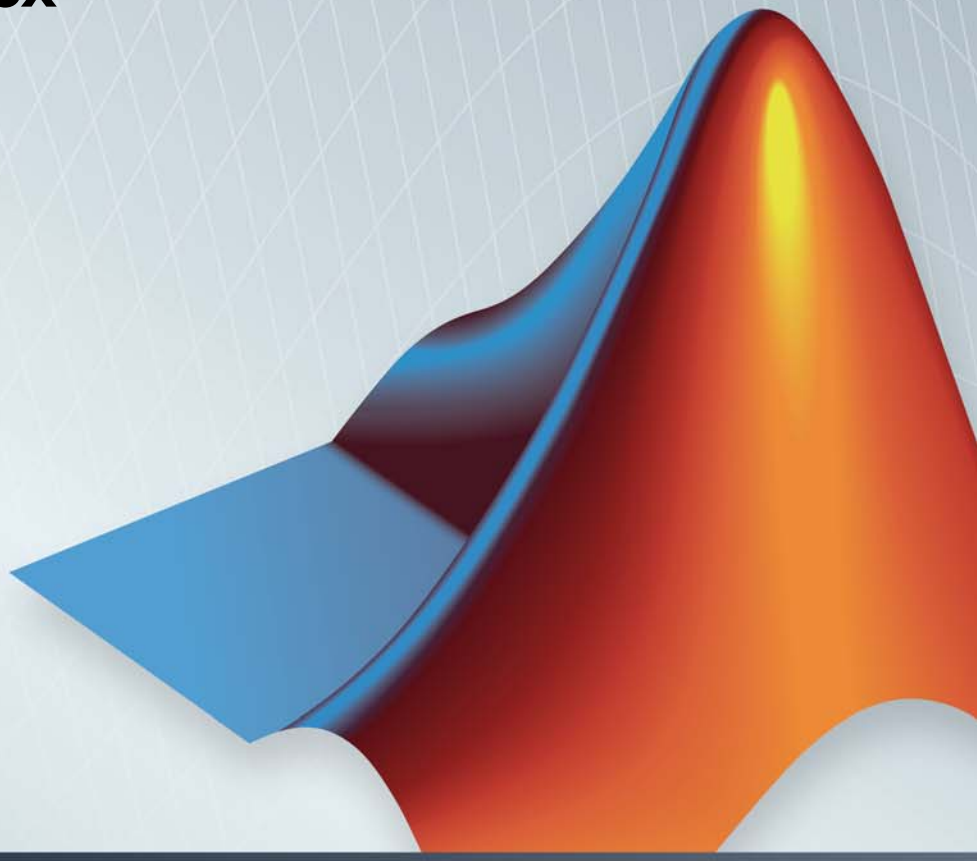
Trading Toolbox™

User's Guide

R2014**a**



MATLAB®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Trading Toolbox™ User's Guide

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

March 2013	Online only	New for Version 1.0 (Release 2013a)
September 2013	Online only	Revised for Version 2.0 (Release 2013b)
March 2014	Online only	Revised for Version 2.1 (Release 2014a)

Contents

1 Getting Started

2 Workflow Models

Sample Code for Workflows

3

4 Functions — Alphabetical List

1

Getting Started

Trading Toolbox Product Description

Access prices and send orders to trading systems

Trading Toolbox™ provides functions for accessing trade and quote pricing data, defining order types, and sending orders to financial trading markets. The toolbox lets you integrate streaming and event-based data into MATLAB®, enabling you to develop financial trading strategies and algorithms that analyze and react to the market in real time. You can build algorithmic or automated trading strategies that work across multiple asset classes, instrument types, and trading markets while integrating with industry-standard trade execution platforms.

With Trading Toolbox, you can subscribe to streams of tradable instrument data, including quotes, volumes, trades, market depth, and instrument metadata. You also can define order types and instructions for how to route and fill orders. Trading Toolbox supports Bloomberg® EMSX, CQG® Integrated Client, Interactive Brokers® TWS, and Trading Technologies® X_TRADER®.

Key Features

- Access to current, intraday, event-based, and real-time tradable instrument data
- Data filtering by instrument and exchange
- Definable order types and execution instructions
- Bloomberg EMSX order execution
- Trading Technologies X_TRADER instrument pricing and order execution
- CQG Integrated Client instrument pricing, order execution, and historical price retrieval
- Interactive Brokers TWS instrument pricing, order execution, and historical price retrieval

Trading System Providers

In this section...

Supported Providers

This toolbox supports connections to financial trading systems provided by the following corporations:

- Bloomberg EMSX from Bloomberg L.P. (`http://www.bloomberg.com`)

  Note Only the Bloomberg Desktop API is supported.

- CQG (`http://www.cqg.com`)
- IB Trader Workstation℠ from Interactive Brokers (`http://www.interactivebrokers.com`)
- X_TRADER from Trading Technologies (`http://www.tradingtechnologies.com`)

See the MathWorks® Web site for the system requirements for connecting to these trading systems.

Connection Requirements

To connect to these trading systems, additional requirements apply. The following data service providers require you to install proprietary software on your PC:

- Bloomberg EMSX

Note You need the Bloomberg Desktop software license for the host on which Trading Toolbox and MATLAB software are running.

- CQG
- Interactive Brokers IB Trader Workstation
- Trading Technologies X_TRADER

You must have a valid license for required client software on your machine.

For more information about how to obtain required software, contact your trading system sales representative.

Create an Order Using Interactive Brokers with MATLAB

This example shows how to connect to the IB Trader Workstation, retrieve historical data, create a market order, and specify a different instrument.

Run the IB Trader Workstation application.

Ensure the IB Trader Workstation application is running, and that API connections are enabled. You can do this from within IB Trader Workstation.

1. Select **File > Global Configuration** to open the Trader Workstation Configuration (Simulated Trading) dialog box.
2. Select **API > Settings**.
3. Ensure that the **Enable ActiveX and Socket Clients** check box is selected.

Connect to the IB Trader Workstation.

Connect to the IB Trader Workstation and create connection `ib` using the local host and default port number `7496`.

```
ib = ibtws('',7496);
```

When the `Accept incoming connection attempt` message appears in the IB Trader Workstation, click **Yes**.

Retrieve historical and current data.

Create the IB Trader Workstation `IContract` object `ibContract`. This object denotes the security. For this example, get data for Microsoft® `MSFT` stock. Specifying `SMART` as the exchange lets Interactive Brokers determine which venues to get data from. Setting the currency type to `USD` clarifies that you want dollar-denominated stock. This is useful when stocks are dual-listed or multi-listed across different jurisdictions.

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'MSFT';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
```

```
ibContract.currency = 'USD';
```

Define the period for which you need data, for example, the last 20 business days, excluding today.

```
bizDayConvention = 13; % i.e. BUS/252
startDate = daysadd(today,-20,bizDayConvention);
endDate   = daysadd(today,-1, bizDayConvention);
```

This code uses the `daysadd` function from Financial Toolbox™ to compute the appropriate start and end dates.

Retrieve historical data for the last 20 business days.

```
histTradeData = history(ib,ibContract,startDate,endDate);
```

Note The `history` function accepts additional parameters that let you obtain other historical data such as option-implied volatility, historical volatility, bid prices, ask prices, or midpoints. If you do not specify anything, the default data returned are last traded prices.

Retrieve current price data from the contract.

```
currentData = getdata(ib, ibContract)
```

```
currentData =

    LAST_PRICE: 34.93
     LAST_SIZE: 1
        VOLUME: 66113
     BID_PRICE: 34.92
      BID_SIZE: 157
     ASK_PRICE: 34.93
      ASK_SIZE: 129
```

Create a trade market order.

The IB Trader Workstation supports a variety of order types, including basic types such as limit orders, stop orders, and market orders. For this example,

set up a stock contract for Microsoft stock. After setting the order type as `MKT`, then specify the action, in this case `BUY`, and the total quantity to trade.

```
ibMktOrder                  = ib.Handle.createOrder;
ibMktOrder.action           = 'BUY';
ibMktOrder.totalQuantity    = 100;
ibMktOrder.orderType        = 'MKT';
```

Set a unique order identifier, and send the orders to Interactive Brokers.

```
ibOrderID = 1;
result = createOrder(ib, ibContract, ibMktOrder, ibOrderID)
```

```
result =

             STATUS: 'Filled'
             FILLED: 100
          REMAINING: 0
     AVG_FILL_PRICE: 34.93
           PERM_ID: '456471585'
          PARENT_ID: 0
    LAST_FILL_PRICE: 34.93
          CLIENT_ID: 0
           WHY_HELD: ''
```

Specify a different instrument.

You can trade a variety of instruments using the IB Trader Workstation API, including equities, futures, options, futures options, and foreign currencies. Here, use the E-mini Standard and Poor's 500 futures contract on the CME Globex with a December 2013 expiry. Specify the symbol as `ES`, the security type to be a futures contract `FUT`, the expiry in a `YYYYMM` date format, the exchange as `GLOBEX`, and the currency as `USD`.

```
ibFutures           = ib.Handle.createContract;
ibFutures.symbol    = 'ES';
ibFutures.secType   = 'FUT';
ibFutures.expiry    = '201312'; % Dec 2013
ibFutures.exchange  = 'GLOBEX';
ibFutures.currency  = 'USD';
```

Close the connection.

After retrieving data and sending orders, close the IB Trader Workstation connection `ib`.

```
close(ib);
```

See Also

`ibtws` | `close` | `history` | `getData` | `createOrder`

External Web Sites

- `http://www.interactivebrokers.com/en/software/api/api.htm`

Create an Order Using CQG with MATLAB

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)
  c.Handle.registerevent({streamEventNames{i},...
                          @(varargin)cqgrealtimeeventhandler(varargin{:})});
end

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

Subscribe to the instrument.

Subscribe to a security tied to the EURIBOR.

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

Create the `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 the 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 the connection.

```
close(c);
```

See Also

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

External Web Sites

- `http://cqg.com/Products/CQG-API/CQG-Trader-API.aspx`

Create an Order Using Bloomberg EMSX with MATLAB

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 *Bloomberg API Developer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.

Connect to Bloomberg EMSX.

1. If you have not used the `emsx` function before, then add the file `blpapi3.jar` to the MATLAB Java® class path. Use the `javaaddpath` function or edit your `javaclasspath.txt` file.

   Note 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`
   - `..\blp\api\APIv3\JavaAPI\v3.3.1.0\lib\blpapi3.jar`

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

   a. In your Bloomberg terminal, type `WAPI {GO}` to display the **Desktop/Server API** screen.

   b. Select **SDK Download Center** and then click **Desktop v3.x API**.

   c. Once you have `blpapi3.jar` on your system, add it to the MATLAB Java class path using `javaaddpath`. This must be done for every session of MATLAB. To avoid repeating this at every session, you can add `javaaddpath` to your `startup.m` file or you can add the full path for `blpapi3.jar` to your `javaclasspath.txt` file. For more information, see "Bringing Java Classes into MATLAB Workspace".

   ---

2. Connect to the Bloomberg EMSX data server.

```
c = emsx(servicename)
```

```
c =

 emsx with properties:

      Session:
      Service:
    Ipaddress:
         Port:
```

`servicename` is a string. The available services are:

- Bloomberg EMSX test service is `'//blp/emapisvc_beta'`.
- Bloomberg EMSX production service is `'//bmp/emapisvc'`.

When you create a Bloomberg EMSX connection using `emsx`, the connection object properties are returned.

```
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
```

Create the request structure.

Set up a market order of 400 shares of IBM® as a structure `reqStruct`.

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

Create the market order and route.

```
rCreateOrderAndRoute = createOrderAndRoute(c,reqStruct);
```

Close the Bloomberg EMSX connection.

```
close(c);
```

See Also

`emsx` | `createOrderAndRoute` | `close`

Create an Order Using Trading Technologies X_TRADER with MATLAB

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 an instrument for a 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 an event handler for the 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 an 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.

```
c.OrderSet(1).registerevent({'OnOrderFilled',...
                             @(varargin)ttorderevent(varargin{:},c)});
c.OrderSet(1).registerevent({'OnOrderRejected',...
                             @(varargin)ttorderevent(varargin{:},c)});
c.OrderSet(1).registerevent({'OnOrderSubmitted',...
                             @(varargin)ttorderevent(varargin{:},c)});
c.OrderSet(1).registerevent({'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 an order profile with the existing instrument.

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

Set the customer default property.

Assign the customer defaults for trading an instrument.

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

Set up the order profile as a 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 the order server status.

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

Verify the order server availability and submit the 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 the connection.

```
close(c);
```

See Also

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

External Web Sites

- `https://developer.tradingtechnologies.com/x_trader-api`

2

Workflow Models

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. Subscribe to orders and routes to obtain events on subsequent requests to Bloomberg EMSX for orders and routes.

   Use the `orders` and `routes` functions.
3. Create a Bloomberg EMSX order. Options in the flow of creating an order are:
   - Create an order using `createOrder`.
   - Create an order and route using `createOrderAndRoute`. Or get route information using `getRouteInfo` and then create an order and route using `createOrderAndRoute`.
   - Create an order and route that uses a strategy with `createOrderAndRouteWithStrat`.
4. Modify an order, or modify the route. Options in the flow of modifying an order are:
   - Modify an order using `modifyOrder`.
   - Modify a route with a strategy using `modifyRouteWithStrat`.
   - Modify a route using `modifyRoute`.
5. Delete the order, or delete a route. Options in the flow of deleting an order are:
   - Delete the order using `deleteOrder`.
   - Delete a route using `deleteRoute`.
6. Manage open order status. Options in the flow of managing order status are:
   - Obtain order information using `getOrderInfo`.

- Obtain route information using `getRouteInfo`.
- Obtain broker information using `getBrokerInfo`.

7 Close the Bloomberg EMSX connection using `close`.

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

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:

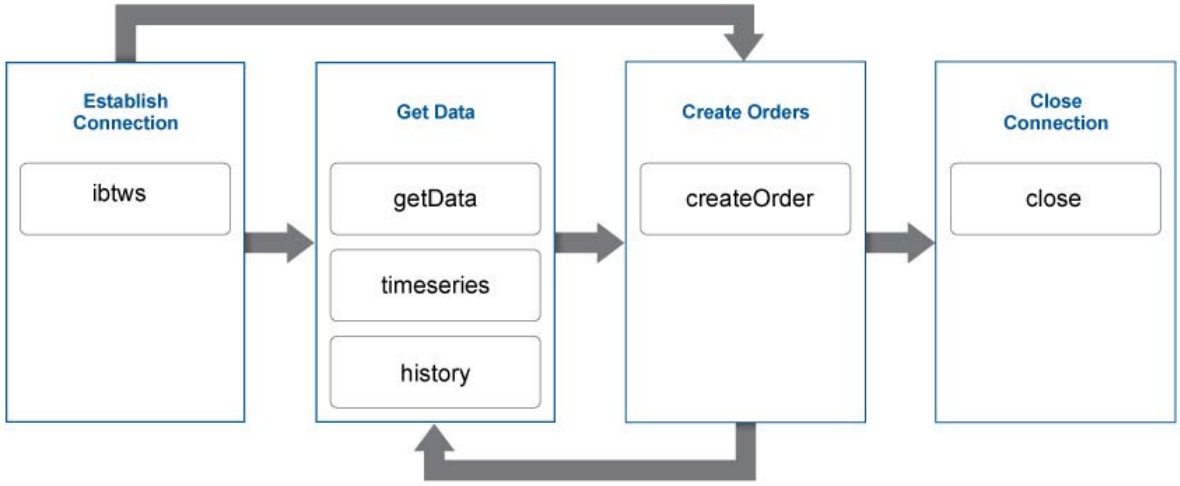
1. Connect to Trading Technologies X_TRADER using `xtrdr`.
2. Create an event notifier using `createNotifier`.
3. Create an instrument and attach it to the notifier using `createInstrument`. Use `getData` to return information on the instrument that you have created.
4. Define events by assigning callbacks 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`.

Related Examples

- "X_TRADER Price Update" on page 3-3
- "X_TRADER Price Update Depth" on page 3-5
- "X_TRADER Order Submission" on page 3-9

Workflow for Interactive Brokers

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



To request current, intraday, or historical data:

1. Connect to IB Trader Workstation using `ibtws`.
2. Create the IB Trader Workstation contract object.
3. Request current data for a security using `getData`.
4. Request intraday data for a security using `timeseries`.
5. Request historical data for a security using `history`.
6. Close the IB Trader Workstation connection using `close`.

To submit orders to IB Trader Workstation:

1. Connect to IB Trader Workstation using `ibtws`.
2. Create the IB Trader Workstation contract object.

3 Create the IB Trader Workstation order object.

4 Create and submit the order using `createOrder`.

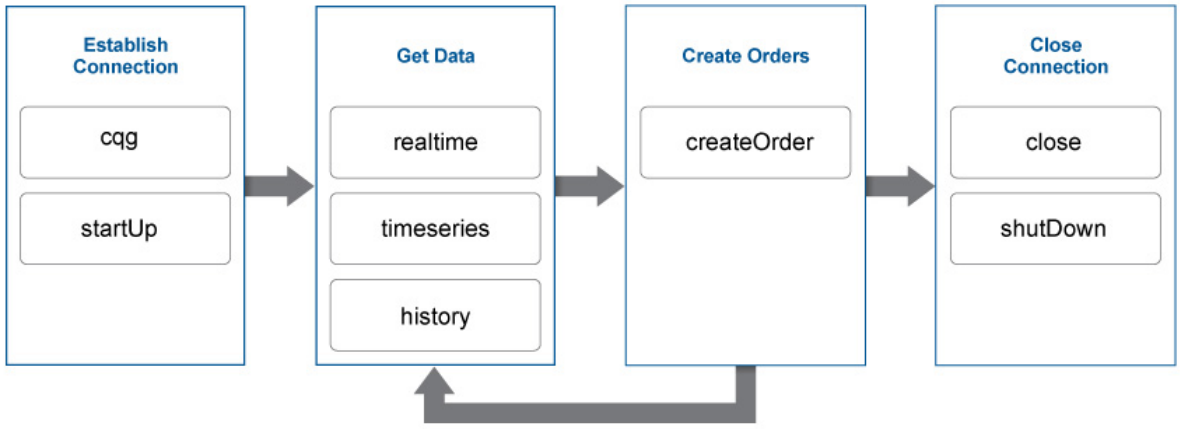
5 Close the IB Trader Workstation connection using `close`.

Related Examples

- "Create Interactive Brokers Order" on page 3-30
- "Request Interactive Brokers Historical Data" on page 3-34
- "Stream Interactive Brokers Data" on page 3-36

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`.

Related Examples

- "Create CQG Order" on page 3-40
- "Request CQG Historical Data" on page 3-46
- "Request CQG Intraday Tick Data" on page 3-49
- "Request CQG Real-Time Data" on page 3-54

3

Sample Code for Workflows

X_TRADER Workflows

X_TRADER supports the following workflows:

X_TRADER Price Update

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.

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

- "X_TRADER Price Update Depth" on page 3-5
- "X_TRADER Order Submission" on page 3-9

Concepts

- "Workflows for Trading Technologies X_TRADER" on page 2-4

X_TRADER Price Update Depth

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,'Exhange','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.

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

Set up the figure window.

Set up the figure window to display depth data.

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

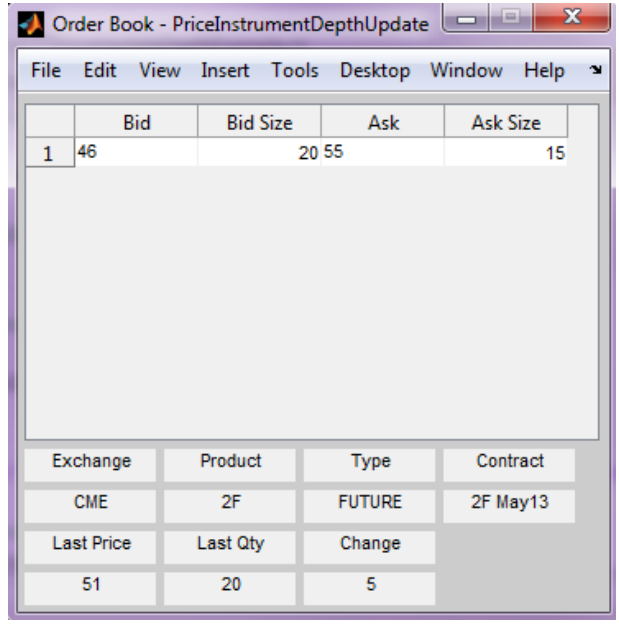
Store data.

```
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

- "X_TRADER Price Update" on page 3-3
- "X_TRADER Order Submission" on page 3-9

Concepts

- "Workflows for Trading Technologies X_TRADER" on page 2-4

X_TRADER Order Submission

This example shows how to connect to X_TRADER and submit an order.

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;
X.Gate.registerevent({'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.

```
X.OrderSet(1).registerevent({'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

- "X_TRADER Price Update" on page 3-3
- "X_TRADER Price Update Depth" on page 3-5

Concepts

- "Workflows for Trading Technologies X_TRADER" on page 2-4

Bloomberg EMSX Workflows

Bloomberg EMSX supports the following workflows:

Bloomberg EMSX Order Management

This example shows how to connect to a Bloomberg EMSX service, set up an order subscription, and create and manage an order.

Connect to Bloomberg EMSX.

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

SessionConnectionUp = {
    server = localhost/127.0.0.1:8194
}

SessionStarted = {
}

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

Set up the order subscription.

```
r = b.orders({'EMSX_TICKER','EMSX_AMOUNT','EMSX_FILL'})

r =

                    MSG_TYPE: {'E'}
                MSG_SUB_TYPE: {'O'}
                EVENT_STATUS: 4
                  API_SEQ_NUM: 1
               EMSX_SEQUENCE: 342481
               EMSX_ROUTE_ID: 0
                EMSX_FILL_ID: 0
                   EMSX_SIDE: {''}
                 EMSX_AMOUNT: 300
                 EMSX_FILLED: 0
              EMSX_AVG_PRICE: 0
                 EMSX_BROKER: {''}
                EMSX_WORKING: 0
                 EMSX_TICKER: {'IBM US Equity'}
```

```
                ...
```

Create the request structure.

Create the request for the specific buy order for IBM stock.

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

% For limit orders, limit price can be set
% reqStruct.EMSX_LIMIT_PRICE = 150;
```

Create the order.

Create a new order.

```
rCreateOrder = b.createOrder(reqStruct)
```

```
rCreateOrder =

    EMSX_SEQUENCE: 344700
          MESSAGE: 'Order created'
```

Get the order status.

```
b.getOrderInfo(344700)
```

```
rOrderStatus1 =

              EMSX_TICKER: 'IBM'
            EMSX_EXCHANGE: 'US'
                EMSX_SIDE: 'BUY'
            EMSX_POSITION: 'BUY'
            EMSX_PORT_MGR: 'CF'
              EMSX_TRADER: 'CF'
```

```
              EMSX_NOTES: ''
             EMSX_AMOUNT: 400
        EMSX_IDLE_AMOUNT: 0
            EMSX_WORKING: 200
             EMSX_FILLED: 200
          EMSX_TS_ORDNUM: 200
        EMSX_LIMIT_PRICE: 0
          EMSX_AVG_PRICE: 189.5900
               EMSX_FLAG: 2
            EMSX_SUB_FLAG: 0
         EMSX_YELLOW_KEY: 'Equity'
        EMSX_BASKET_NAME: ''
  EMSX_ORDER_CREATE_DATE: '12/06/12'
  EMSX_ORDER_CREATE_TIME: '14:28:37'
         EMSX_ORDER_TYPE: 'MKT'
                EMSX_TIF: 'DAY'
             EMSX_BROKER: 'BB'
        EMSX_TRADER_UUID: '1244972'
    EMSX_STEP_OUT_BROKER: ''
```

Modify the order.

Change the properties for an existing order and then route the order.

```
modStruct.EMSX_SEQUENCE = rCreateOrder.EMSX_SEQUENCE;
modStruct.EMSX_TICKER = 'IBM';
modStruct.EMSX_AMOUNT = int32(300);
rModifyOrder = b.modifyOrder(modStruct)

%Route order
% routeStruct.EMSX_AMOUNT = modStruct.EMSX_AMOUNT;
% routeStruct.EMSX_SEQUENCE = rModifyOrder.EMSX_SEQUENCE;
% routeStruct.EMSX_TICKER = reqStruct.EMSX_TICKER;
% routeStruct.EMSX_ORDER_TYPE = reqStruct.EMSX_ORDER_TYPE;
% routeStruct.EMSX_BROKER = reqStruct.EMSX_BROKER;
% routeStruct.EMSX_TIF = reqStruct.EMSX_TIF;
% routeStruct.EMSX_HAND_INSTRUCTION = reqStruct.EMSX_HAND_INSTRUCTION;
% routeStruct.EMSX_ODD_LOT = '-1';
% routeStruct.EMSX_CFD_FLAG = '-1';
% routeStruct.EMSX_RELEASE_TIME = '-1';
```

```
% rRouteOrder = b.routeOrder(routeStruct);

rModifyOrder =

    EMSX_SEQUENCE: 344700
          MESSAGE: 'Order Modified'
```

Get the modified order status.

```
rOrderStatus2 = b.getOrderInfo(344700)

rOrderStatus2 =

              EMSX_TICKER: 'IBM'
            EMSX_EXCHANGE: 'US'
                EMSX_SIDE: 'BUY'
            EMSX_POSITION: 'BUY'
            EMSX_PORT_MGR: 'CF'
              EMSX_TRADER: 'CF'
               EMSX_NOTES: ''
              EMSX_AMOUNT: 300
         EMSX_IDLE_AMOUNT: 0
             EMSX_WORKING: 200
              EMSX_FILLED: 100
           EMSX_TS_ORDNUM: 200
         EMSX_LIMIT_PRICE: 0
           EMSX_AVG_PRICE: 189.5900
                EMSX_FLAG: 2
            EMSX_SUB_FLAG: 0
          EMSX_YELLOW_KEY: 'Equity'
         EMSX_BASKET_NAME: ''
    EMSX_ORDER_CREATE_DATE: '12/06/12'
    EMSX_ORDER_CREATE_TIME: '14:28:37'
          EMSX_ORDER_TYPE: 'MKT'
                 EMSX_TIF: 'DAY'
              EMSX_BROKER: 'BB'
       EMSX_TRADER_UUID: '1244972'
  EMSX_STEP_OUT_BROKER: ''
```

Delete the order (if necessary).

The structure returned from the `createOrder` call can be used as the input to delete the order or you can create a new structure where the field `EMSX_SEQUENCE` contains the order number to be canceled.

```
delStruct.EMSX_SEQUENCE = rCreateOrder.EMSX_SEQUENCE;
rDeleteOrder = b.deleteOrder(delStruct)
```

```
rDeleteOrder =

     STATUS: 'O'
    MESSAGE: 'Order deleted'
```

Close the connection.

```
close(b)
processEvent(b)
```

```
SessionConnectionDown = {
    server = localhost/127.0.0.1:8194
}
```

See Also

`createOrder` | `orders` | `modifyOrder` | `deleteOrder` | `routeOrder`

Related Examples

- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

Bloomberg EMSX Route Management

This example shows how to connect to a Bloomberg EMSX service, set up a route subscription, and create and manage a route.

Connect to Bloomberg EMSX.

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

SessionConnectionUp = {
    server = localhost/127.0.0.1:8194
}

SessionStarted = {
}

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

Set up the route subscription.

```
rRouteStatus1 = b.routes({'EMSX_BROKER','EMSX_WORKING'})

rRouteStatus1 =

                    MSG_TYPE: {4x1 cell}
                MSG_SUB_TYPE: {4x1 cell}
                EVENT_STATUS: [4x1 int32]
                  API_SEQ_NUM: [4x1 int64]
               EMSX_SEQUENCE: [4x1 int32]
               EMSX_ROUTE_ID: [4x1 int32]
                EMSX_FILL_ID: [4x1 int32]
                   EMSX_SIDE: {4x1 cell}
                 EMSX_AMOUNT: [4x1 int32]
                 EMSX_FILLED: [4x1 int32]
              EMSX_AVG_PRICE: [4x1 double]
                 EMSX_BROKER: {4x1 cell}
                EMSX_WORKING: [4x1 int32]
                 EMSX_TICKER: {4x1 cell}
```

```
                    ...
```

Create the request structure.

Create the request for a specific buy order for IBM stock.

```
reqStruct.EMSX_TICKER = 'IBM';
reqStruct.EMSX_AMOUNT = int32(3358);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';

% create and route order
rOrder = b.createOrderAndRoute(reqStruct)

rOrder =

    EMSX_SEQUENCE: 348930
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

Check the route status:

```
routeStruct.EMSX_SEQUENCE = rOrder.EMSX_SEQUENCE
routeStruct.EMSX_ROUTE_ID = rOrder.EMSX_ROUTE_ID

rRouteStatus2 = b.getRouteInfo(routeStruct)

rRouteStatus2 =

                EMSX_AVG_PRICE: 189.5900
                    EMSX_YIELD: 0
         EMSX_ROUTE_CREATE_DATE: 20121206
         EMSX_ROUTE_CREATE_TIME: 142837
    EMSX_ROUTE_LAST_UPDATE_DATE: 20121206
    EMSX_ROUTE_LAST_UPDATE_TIME: 142838
            EMSX_SETTLE_DATE: 20121211
                 EMSX_AMOUNT: 400
```

```
              EMSX_FILLED: 200
     EMSX_IS_MANUAL_ROUTE: 0
              EMSX_BROKER: 'BB'
             EMSX_ACCOUNT: ''
           EMSX_STATUS_ID: 51088
              EMSX_STATUS: 'PtlFil'
    EMSX_HAND_INSTRUCTION: 'ANY'
          EMSX_ORDER_TYPE: 'MKT'
                 EMSX_TIF: 'DAY'
              EMSX_LOC_ID: ''
          EMSX_LOC_BROKER: 'DAY'
          EMSX_STOP_PRICE: 0
        EMSX_BLOT_SEQ_NUM: 1
           EMSX_BLOT_DATE: 20121206
           EMSX_COMM_TYPE: 'DAY'
           EMSX_COMM_RATE: 0
    EMSX_USER_COMM_AMOUNT: 0
           EMSX_LSTTR2ID0: 1.3548e+09
           EMSX_LSTTR2ID1: 284950536
         EMSX_LIMIT_PRICE: 0
```

Modify the route.

Modify the properties for the previously created route.

```
modStruct.EMSX_SEQUENCE = rOrder.EMSX_SEQUENCE;
modStruct.EMSX_ROUTE_ID = rOrder.EMSX_ROUTE_ID;
modStruct.EMSX_TICKER = 'IBM';
modStruct.EMSX_AMOUNT = int32(3000);
modStruct.EMSX_ORDER_TYPE = 'MKT';
modStruct.EMSX_TIF = 'DAY';
rModifyRoute = b.modifyRoute(modStruct);
```

Check the route status for the modified route.

```
rRouteStatus3 = b.getRouteInfo(routeStruct)
```

```
rRouteStatus3 =

                EMSX_AVG_PRICE: 189.7900
                    EMSX_YIELD: 0
```

```
     EMSX_ROUTE_CREATE_DATE: 20121206
     EMSX_ROUTE_CREATE_TIME: 142837
EMSX_ROUTE_LAST_UPDATE_DATE: 20121206
EMSX_ROUTE_LAST_UPDATE_TIME: 143251
           EMSX_SETTLE_DATE: 20121211
                EMSX_AMOUNT: 250
                EMSX_FILLED: 250
       EMSX_IS_MANUAL_ROUTE: 0
                EMSX_BROKER: 'BB'
               EMSX_ACCOUNT: ''
             EMSX_STATUS_ID: 199032
                EMSX_STATUS: 'Filled'
      EMSX_HAND_INSTRUCTION: 'ANY'
            EMSX_ORDER_TYPE: 'MKT'
                   EMSX_TIF: 'DAY'
                EMSX_LOC_ID: ''
            EMSX_LOC_BROKER: 'DAY'
            EMSX_STOP_PRICE: 0
          EMSX_BLOT_SEQ_NUM: 1
             EMSX_BLOT_DATE: 20121206
             EMSX_COMM_TYPE: 'DAY'
             EMSX_COMM_RATE: 0
      EMSX_USER_COMM_AMOUNT: 0
             EMSX_LSTTR2ID0: 1.3548e+09
             EMSX_LSTTR2ID1: 284950536
          EMSX_LIMIT_PRICE: 0
```

Delete the route.

The structure returned from the `createOrderAndRoute` call can be used as the input to delete the route or you can create a new structure where the field `EMSX_SEQUENCE` contains the order number to be canceled.

```
delStruct.EMSX_SEQUENCE = rOrder.EMSX_SEQUENCE;
delStruct.EMSX_ROUTE_ID = rOrder.EMSX_ROUTE_ID;
rDeleteRoute = b.deleteRoute(delStruct)
```

```
rDeleteRoute =

    STATUS: '0'
```

```
     MESSAGE: 'Route deleted'
```

Close the connection.

```
close(b)
processEvent(b)

SessionConnectionDown = {
    server = localhost/127.0.0.1:8194
}
```

See Also

`createOrderAndRoute` | `modifyRoute` | `deleteRoute` | `routes` | `routeOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

Bloomberg EMSX Order and Route Management

This example shows how to connect to a Bloomberg EMSX service, set up an order and route subscription, and create and manage an order and route.

Connect to Bloomberg EMSX.

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

SessionConnectionUp = {
    server = localhost/127.0.0.1:8194
}

SessionStarted = {
}

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

Set up the order and route subscription.

```
o = b.orders({'EMSX_TICKER','EMSX_AMOUNT','EMSX_FILL'})
r = b.routes({'EMSX_BROKER','EMSX_WORKING'})

o =

                    MSG_TYPE: {4x1 cell}
                MSG_SUB_TYPE: {4x1 cell}
                EVENT_STATUS: [4x1 int32]
                  API_SEQ_NUM: [4x1 int64]
               EMSX_SEQUENCE: [4x1 int32]
               EMSX_ROUTE_ID: [4x1 int32]
                EMSX_FILL_ID: [4x1 int32]
                   EMSX_SIDE: {4x1 cell}
                 EMSX_AMOUNT: [4x1 int32]
                 EMSX_FILLED: [4x1 int32]
             EMSX_AVG_PRICE: [4x1 double]
                EMSX_BROKER: {4x1 cell}
               EMSX_WORKING: [4x1 int32]
```

```
                    EMSX_TICKER: {4x1 cell}
                  EMSX_EXCHANGE: {4x1 cell}

                  ...

r =

                       MSG_TYPE: {2x1 cell}
                   MSG_SUB_TYPE: {2x1 cell}
                   EVENT_STATUS: [2x1 int32]
                    API_SEQ_NUM: [2x1 int64]
                  EMSX_SEQUENCE: [2x1 int32]
                  EMSX_ROUTE_ID: [2x1 int32]
                   EMSX_FILL_ID: [2x1 int32]
                      EMSX_SIDE: {2x1 cell}
                    EMSX_AMOUNT: [2x1 int32]
                    EMSX_FILLED: [2x1 int32]
                 EMSX_AVG_PRICE: [2x1 double]
                    EMSX_BROKER: {2x1 cell}
                   EMSX_WORKING: [2x1 int32]
                    EMSX_TICKER: {2x1 cell}

                  ...
```

Create the request structure.

Create a request for a specific buy order for IBM stock.

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

%For Limit orders, limit price can be set
%reqStruct.EMSX_LIMIT_PRICE = 150;
```

Create the order and route.

Create an order and route for execution.

```
rCreateOrderAndRoute = b.createOrderAndRoute(reqStruct)
```

```
rCreateOrderAndRoute =

    EMSX_SEQUENCE: 344705
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order created and routed'
```

Get the order status.

```
rOrderStatus1 = b.getOrderInfo(rCreateOrderAndRoute.EMSX_SEQUENCE)
```

```
rOrderStatus1 =

              EMSX_TICKER: 'IBM'
            EMSX_EXCHANGE: 'US'
                EMSX_SIDE: 'BUY'
            EMSX_POSITION: 'BUY'
            EMSX_PORT_MGR: 'CG'
              EMSX_TRADER: 'CG'
               EMSX_NOTES: ''
              EMSX_AMOUNT: 400
         EMSX_IDLE_AMOUNT: 0
             EMSX_WORKING: 200
              EMSX_FILLED: 200
           EMSX_TS_ORDNUM: 200
         EMSX_LIMIT_PRICE: 0
           EMSX_AVG_PRICE: 189.5900
                EMSX_FLAG: 2
            EMSX_SUB_FLAG: 0
          EMSX_YELLOW_KEY: 'Equity'
        EMSX_BASKET_NAME: ''
EMSX_ORDER_CREATE_DATE: '12/06/12'
EMSX_ORDER_CREATE_TIME: '14:28:37'
       EMSX_ORDER_TYPE: 'MKT'
               EMSX_TIF: 'DAY'
            EMSX_BROKER: 'BB'
        EMSX_TRADER_UUID: '1244972'
```

Get the route status.

```
routeStat.EMSX_SEQUENCE = rCreateOrderAndRoute.EMSX_SEQUENCE
routeStat.EMSX_ROUTE_ID = rCreateOrderAndRoute.EMSX_ROUTE_ID

rRouteStatus1 = b.getRouteInfo(routeStat)
```

```
rRouteStatus1 =

                EMSX_AVG_PRICE: 189.5900
                    EMSX_YIELD: 0
         EMSX_ROUTE_CREATE_DATE: 20121206
         EMSX_ROUTE_CREATE_TIME: 142837
    EMSX_ROUTE_LAST_UPDATE_DATE: 20121206
    EMSX_ROUTE_LAST_UPDATE_TIME: 142838
            EMSX_SETTLE_DATE: 20121211
                 EMSX_AMOUNT: 400
                 EMSX_FILLED: 200
          EMSX_IS_MANUAL_ROUTE: 0
                 EMSX_BROKER: 'BB'
                EMSX_ACCOUNT: ''
               EMSX_STATUS_ID: 51088
                  EMSX_STATUS: 'PtlFil'
         EMSX_HAND_INSTRUCTION: 'ANY'
               EMSX_ORDER_TYPE: 'MKT'
                     EMSX_TIF: 'DAY'
                  EMSX_LOC_ID: ''
               EMSX_LOC_BROKER: 'DAY'
               EMSX_STOP_PRICE: 0
             EMSX_BLOT_SEQ_NUM: 1
                EMSX_BLOT_DATE: 20121206
                EMSX_COMM_TYPE: 'DAY'
                EMSX_COMM_RATE: 0
         EMSX_USER_COMM_AMOUNT: 0
                EMSX_LSTTR2ID0: 1.3548e+09
                EMSX_LSTTR2ID1: 284950536
              EMSX_LIMIT_PRICE: 0
```

Modify the order on route.

Modify the previously routed order.

```
modStruct.EMSX_SEQUENCE = rCreateOrderAndRoute.EMSX_SEQUENCE;
modStruct.EMSX_ROUTE_ID = rCreateOrderAndRoute.EMSX_ROUTE_ID;
modStruct.EMSX_TICKER = 'IBM';
modStruct.EMSX_AMOUNT = int32(250);
modStruct.EMSX_ORDER_TYPE = 'MKT';
modStruct.EMSX_TIF = 'DAY';
rModifyRoute = b.modifyRoute(modStruct)
```

```
rModifyRoute =

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

Delete order.

The structure returned from the `createOrderAndRoute` call can be used as the input to delete the order or you can create a new structure where the field `EMSX_SEQUENCE` contains the order number to be canceled.

```
delStruct.EMSX_SEQUENCE = rCreateOrderAndRoute.EMSX_SEQUENCE;
delStruct.EMSX_ROUTE_ID = rCreateOrderAndRoute.EMSX_ROUTE_ID;
rDeleteOrder = b.deleteOrder(delStruct)
```

```
rDeleteOrder =

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

Close the connection.

```
close(b)
processEvent(b)
```

```
SessionConnectionDown = {
    server = localhost/127.0.0.1:8194
}
```

See Also

`createOrderAndRoute` | `orders` | `modifyOrder` | `deleteOrder` | `routes` | `routeOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

Create Interactive Brokers Order

This example shows how to connect to the IB Trader Workstation, define event handlers, create an `IContract` object, create an `IOrder` object, and execute the order.

Connect to IB Trader Workstation.

Connect to IB Trader Workstation and create connection `ib` using the local host and port number `7496`.

```
ib = ibtws('',7496);
```

Register an event handler to track error and order status events.

```
eventNames = {'errMsg','orderStatus'};
for i = 1:length(eventNames)
  registerevent(ib.Handle,{eventNames{i},...
      @(varargin)ibExampleOrderEventHandler(varargin{:})})
end
```

The example event handler `ibExampleOrderEventHandler` is assigned to the events in `eventNames`.

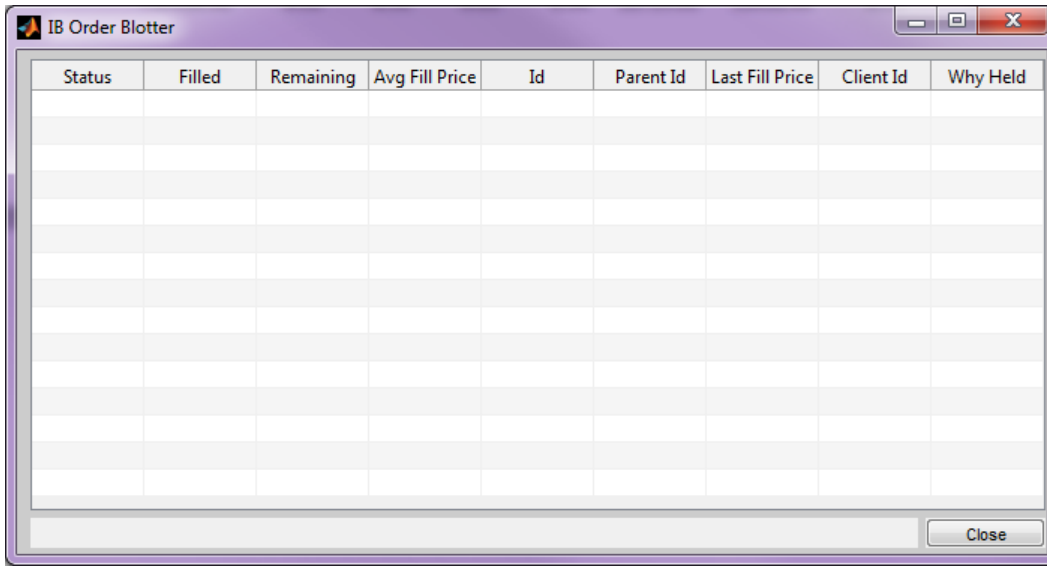
Create example order blotter.

Create an example order blotter to be populated by the event handler.

This MATLAB code creates a MATLAB figure to contain the Interactive Brokers order information once the order is placed.

```
f = findobj('Tag','IBOrderBlotter');
if isempty(f)
  f = figure('Tag','IBOrderBlotter','MenuBar','none',...
     'NumberTitle','off','Name','IB Order Blotter');
  pos = get(f,'Position');
  set(f,'Position',[pos(1) pos(2) 687 335])
  colnames = {'Status' 'Filled' 'Remaining' 'Avg Fill Price' 'Id' ...
      'Parent Id','Last Fill Price','Client Id','Why Held'};
  data = cell(15,9);
  uitable(f,'Data',data,'RowName',[],'ColumnName',colnames,...
```

```
    'Position',[10 30 677 300],'Tag','OrderDataTable');
  uicontrol('Style','text','Position',[10 5 592 20],...
     'Tag','IBOrderMessage');
  uicontrol('Style','pushbutton','String','Close',...
    'Callback','evalin(''base'',''close(ib);close(findobj(''''Tag''''',''''IBOrderBlotter''''));'')',...
    'Position',[607 5 80 20]);
end
```



MATLAB displays the order blotter figure.

Request information on all open orders.

```
ib.Handle.reqAllOpenOrders
```

Create the IB Trader Workstation contract and order objects.

Create the IB Trader Workstation `IContract` object `ibContract`. This object denotes the security. For example, to create an `IContract` object for symbol `XYZ`, with stock security type, an aggregate exchange, and USD currency, type the following.

```
ibContract = ib.Handle.createContract;
```

```
ibContract.symbol = 'XYZ';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.currency = 'USD'

ibContract =

 Interface.Tws_ActiveX_Control_module.IContract
```

`ibContract` contains the stock symbol, security type, exchange and currency, for security `XYZ`. For more information about creating an `IContract` object, see *Interactive Brokers API Reference Guide*.

Create the IB Trader Workstation `IOrder` object `ibOrder` for a buy market order for two shares.

```
ibOrder = ib.Handle.createOrder;
ibOrder.action = 'BUY';
ibOrder.totalQuantity = 2;
ibOrder.orderType = 'MKT'

ibOrder =

 Interface.Tws_ActiveX_Control_module.IOrder
```

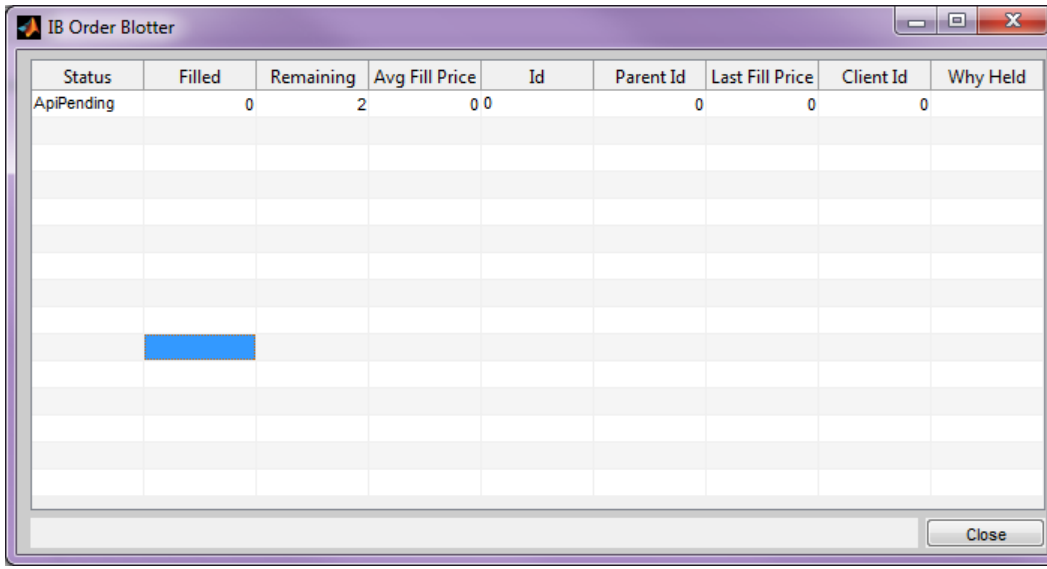
`ibOrder` contains the action, total quantity, and order type. For more information about creating an `IOrder` object, see *Interactive Brokers API Reference Guide*.

Create the Interactive Brokers order.

Execute the buy market order for two shares using a unique order identifier `orderId`.

```
orderId = floor((now-floor(now))*24*3600*1000);

placeOrderEx(ib.Handle,orderId,ibContract,ibOrder)
```



MATLAB displays order information in the order blotter figure.

Close the connection.

Close the IB Trader Workstation connection `ib`.

```
close(ib);
```

See Also

`ibtws` | `close` | `createOrder` | `getData` | `history` | `timeseries`

Related Examples

- "Request Interactive Brokers Historical Data" on page 3-34
- "Stream Interactive Brokers Data" on page 3-36

Concepts

- "Workflow for Interactive Brokers" on page 2-6

External Web Sites

- *Interactive Brokers API Reference Guide*

Request Interactive Brokers Historical Data

This example shows how to connect to IB Trader Workstation, create an `IContract` object, and request historical data.

Connect to IB Trader Workstation and create the IB Trader Workstation contract object.

Connect to IB Trader Workstation and create connection `ib` using the local host and port number `7496`.

```
ib = ibtws('',7496);
```

MATLAB returns `ib` as the connection to IB Trader Workstation with the Interactive Brokers ActiveX® object, the local host, and the chosen port number.

Create the IB Trader Workstation `IContract` object `ibContract`. This object denotes the security. For example, to create an `IContract` object for symbol `XYZ`, with the stock security type, an aggregate exchange, and USD currency, type the following.

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'XYZ';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.currency = 'USD'
```

```
ibContract =

 Interface.Tws_ActiveX_Control_module.IContract
```

`ibContract` contains the stock symbol, security type, exchange, and currency for security `XYZ`. For more information about creating an `IContract` object, see *Interactive Brokers API Reference Guide*.

Request Interactive Brokers historical data.

Request the last 5 days of historical data using `ibContract`.

```
startdate = floor(now) - 5;
```

```
enddate = floor(now);

d = history(ib,ibContract,startdate,enddate)

d =
   1.0e+05 *
    7.3534    0.0079    0.0080    0.0078    0.0078    0.2386    0.1727    0
    7.3534    0.0078    0.0080    0.0078    0.0079    0.1669    0.1075    0
    7.3534    0.0079    0.0079    0.0078    0.0078    0.1982    0.1420    0
    7.3534    0.0079    0.0080    0.0076    0.0078    0.3188    0.2239    0
    7.3534    0.0078    0.0080    0.0077    0.0080    0.5568    0.3723    0
```

`d` contains the historical data for 5 days.

Each row of `d` contains historical data for 1 day. The columns in matrix `d` are a numeric representation of a date, open price, high price, low price, close price, volume, bar count, weighted average price, and flag indicating if there are gaps in the bar.

Close the connection.

Close the IB Trader Workstation connection `ib`.

```
close(ib);
```

See Also

`ibtws` | `close` | `createOrder` | `getData` | `history` | `timeseries`

Related Examples

- "Create Interactive Brokers Order" on page 3-30
- "Stream Interactive Brokers Data" on page 3-36

Concepts

- "Workflow for Interactive Brokers" on page 2-6

External Web Sites

- *Interactive Brokers API Reference Guide*

Stream Interactive Brokers Data

This example shows how to connect to IB Trader Workstation, create `IContract` objects, and stream data.

Connect to IB Trader Workstation and create the real-time data display figure.

Connect to IB Trader Workstation and create connection `ib` using the local host and port number `7496`.

```
ib = ibtws('',7496);
```

MATLAB returns `ib` as the connection to the IB Trader Workstation with the Interactive Brokers ActiveX object, the local host, and the chosen port number.

Register an event handler to track error and tick events for streaming data.

```
eventNames = {'errMsg','tickSize','tickString','tickPrice'};
for i = 1:length(eventNames)
  registerevent(ib.Handle,{eventNames{i},...
      @(varargin)ibExampleRealtimeEventHandler(varargin{:})})
end
```

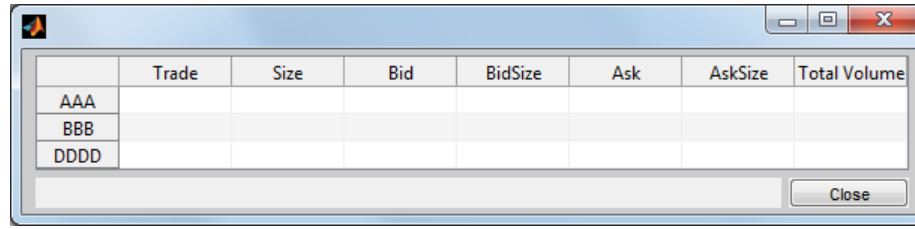
The example event handler `ibExampleRealtimeEventHandler` is assigned to the events in `eventNames`.

Create an example figure to display real-time data.

This MATLAB code creates a MATLAB figure to contain the Interactive Brokers real-time data once the request is made.

```
f = findobj('Tag','IBStreamingDataWorkflow');
if isempty(f)
  f = figure('Tag','IBStreamingDataWorkflow','MenuBar','none',...
     'NumberTitle','off');
  pos = get(f,'Position');
  set(f,'Position',[pos(1) pos(2) pos(3)+37 109])
  colnames = {'Trade' 'Size' 'Bid' 'BidSize' 'Ask' 'AskSize' ...
      'Total Volume'};
  rownames = {'AAA','BBB','DDDD'};
  data = cell(3,6);
```

```
  uitable(f,'Data',data,'RowName',rownames,'ColumnName',colnames,...
  'Position',[10 30 582 76],'Tag','SecurityDataTable');
  uicontrol('Style','text','Position',[10 5 497 20],'Tag','IBMessage');
  uicontrol('Style','pushbutton','String','Close',...
    'Callback',...
    'evalin(''base'',''close(ib);close(findobj(''''Tag'''','''''IBStreamingDataWorkflow'''''));'')',...
    'Position',[512 5 80 20]);
end
```



MATLAB displays the empty figure.

Create IB Trader Workstation contract objects and stream Interactive Brokers data.

Create the IB Trader Workstation `IContract` object `ibContract`. This object denotes the security. For example, create an `IContract` object for symbol `AAA` with the stock security type, an aggregate exchange, and USD currency. Then, request stream data.

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'AAA';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.currency = 'USD';

reqMktDataEx(ib.Handle,1,ibContract,'100',0);
```

`ibContract` contains the stock symbol, security type, exchange, and currency for security `AAA`. For more information about creating an `IContract` object, see *Interactive Brokers API Reference Guide*.

Display the data in the `symbol` property of `ibContract`.

```
ibContract.symbol

ans =
    AAA
```

Create the IB Trader Workstation `IContract` object `ibContract`. This object denotes the security. For example, create an `IContract` object for symbol `BBB` with the stock security type, an aggregate exchange, and USD currency. Then, request stream data.

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'BBB';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.currency = 'USD';

reqMktDataEx(ib.Handle,2,ibContract,'100',0);
```

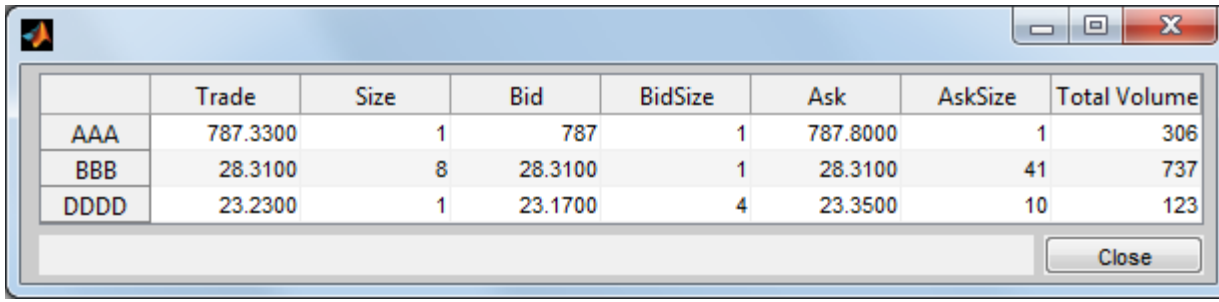
`ibContract` contains the stock symbol, security type, exchange, and currency for security `BBB`.

Create the IB Trader Workstation `IContract` object `ibContract`. This object denotes the security. For example, create an `IContract` object for symbol `DDDD` with the stock security type, an aggregate exchange, and USD currency. Then, request stream data.

```
ibContract = ib.Handle.createContract;
ibContract.symbol = 'DDDD';
ibContract.secType = 'STK';
ibContract.exchange = 'SMART';
ibContract.currency = 'USD';

reqMktDataEx(ib.Handle,3,ibContract,'100',0);
```

`ibContract` contains the stock symbol, security type, exchange, and currency for security `DDDD`.



MATLAB displays the figure populated with real-time data for stock symbols `AAA`, `BBB`, and `DDDD`.

Close the connection.

Close the IB Trader Workstation connection `ib`.

```
close(ib);
```

See Also

`ibtws` | `close` | `createOrder` | `getData` | `history` | `timeseries`

Related Examples

- "Create Interactive Brokers Order" on page 3-30
- "Request Interactive Brokers Historical Data" on page 3-34

Concepts

- "Workflow for Interactive Brokers" on page 2-6

External Web Sites

- *Interactive Brokers API Reference Guide*

Create CQG Order

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 an event handler 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
```

The example event handler `cqgconnectioneventhandler` is assigned to the events in `eventNames`.

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

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

`c.APIConfig` is a CQG configuration object. For more information 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, type 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 more information 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 string 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 more information 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

- "Request CQG Historical Data" on page 3-46
- "Request CQG Real-Time Data" on page 3-54
- "Request CQG Intraday Tick Data" on page 3-49

Concepts

- "Workflow for CQG" on page 2-8

External Web Sites

- *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 an event handler 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
```

The example event handler `cqgconnectioneventhandler` is assigned to the events in `eventNames`.

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

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

`c.APIConfig` is a CQG configuration object. For more information 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`.

```
instrument = 'XYZ.XYZ';
startdate = floor(now) - 10;
enddate = floor(now);
period = 'hpDaily';

history(c,instrument,startdate,enddate,period,x);
```

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 CQG Order" on page 3-40
- "Request CQG Real-Time Data" on page 3-54
- "Request CQG Intraday Tick Data" on page 3-49

Concepts

- "Workflow for CQG" on page 2-8

External Web Sites

- *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 an event handler to track events associated with the connection status.

```
eventNames = {'CELStarted','DataError','IsReady',...
      'DataConnectionStatusChanged'};
for i = 1:length(eventNames)
  c.Handle.registerevent({eventNames{i},...
         @(varargin)cqgconnectioneventhandler(varargin{:})});
end
```

The example event handler `cqgconnectioneventhandler` is assigned to the events in `eventNames`.

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

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

`c.APIConfig` is a CQG configuration object. For more information 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`.

```
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`.

```
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
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2
    0.0007   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2.1475   -2
    ...
```

`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 CQG Order" on page 3-40
- "Request CQG Historical Data" on page 3-46
- "Request CQG Real-Time Data" on page 3-54

Concepts

- "Workflow for CQG" on page 2-8

External Web Sites

- *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 an event handler 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
```

The example event handler `cqgconnectioneventhandler` is assigned to the events in `eventNames`.

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

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

`c.APIConfig` is a CQG configuration object. For more information 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, 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);
```

See Also

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

Related Examples

- "Create CQG Order" on page 3-40
- "Request CQG Historical Data" on page 3-46
- "Request CQG Intraday Tick Data" on page 3-49

Concepts

- "Workflow for CQG" on page 2-8

External Web Sites

- *CQG API Reference Guide*

4

Functions — Alphabetical List

Purpose

Create Bloomberg EMSX connection

Syntax

```
c = emsx(servicename)
```

Description

`c = emsx(servicename)` creates a connection to the local Bloomberg EMSX communications server and uses the service `servicename`.

Input Arguments

servicename - Bloomberg EMSX service name

string

Bloomberg EMSX service name, specified using a test or production Bloomberg EMSX `servicename`.

Data Types

`char`

Output Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, returned as an object structure.

Examples

Connect to the Test Bloomberg EMSX Service

Connect to test calls 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
```

Connect to the Bloomberg EMSX Production Service

Connect to place "live" calls to the Bloomberg EMSX production service.

```
c = emsx('//bmp/emapisvc')

c =

  emsx with properties:

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

See Also

`createOrder` | `createOrderAndRoute` | `close`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

close

Purpose

Close Bloomberg EMSX connection

Syntax

```
close(c)
```

Description

`close(c)` closes a connection to Bloomberg EMSX.

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

Examples

Close the Connection to the Bloomberg EMSX Service

Close connection `c`:

```
close(c)
```

See Also

`emsx`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

Purpose

Create Bloomberg EMSX order

Syntax

```
R = createOrder(c,reqStruct)
R = createOrder(c,reqStruct,Name,Value)
```

Description

`R = createOrder(c,reqStruct)` creates a Bloomberg EMSX order and returns the order sequence number and status message using the default event handler.

`R = createOrder(c,reqStruct,Name,Value)` uses additional options specified by one or more `Name`, `Value` pair arguments. Create a Bloomberg EMSX order using the optional name-value arguments to specify a custom event handler or timeout value for the event handler.

Note Name-value pair arguments can be input as a single input structure containing some or all of the property fields, for example:

```
p.timeOut = 1000;
p.useDefaultEventHandler = false;
createOrder(c,reqStruct,p)
c.processEvent
```

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

reqStruct - Order request structure

structure

Order request structure, specified using EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `reqStruct`.

Example:
```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
```

Data Types
`struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
createOrder(C,reqStruct,'useDefaultEventHandler',false)
```

'useDefaultEventHandler' - Flag for event handler preference

`true` (default) | logical with value `true` or `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, as specified by the string `true` or `false`. When this flag is set to the default, `true`, the default event handler is used. If a custom event handler is used, this flag must be set to `false`.

Example: `'useDefaultEventHandler',false`

Data Types
`logical`

'timeOut' - Connection timeout value for event handler

500 milliseconds (default) | nonnegative integer

Connection timeout value, specified as a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types
`char`

Output Arguments

R - Return for order status

structure

Return for order status, returned as a structure.

Examples

Create Bloomberg EMSX Order Using Default Event Handler

Define the order request structure and create the order.

```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
r = createOrder(c,reqStruct)
```

```
 r =

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

Create Bloomberg EMSX Order Using Custom Event Handler

Define the order request structure and create the order.

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

```
createOrder(c,reqStruct,'useDefaultEventHandler',false)
processEvent(c)

 MESSAGE: 'Order created' CreateOrder = {

     EMSX_SEQUENCE = 354651

     MESSAGE = Order created

     }
```

Create Bloomberg EMSX Order Using `timeOut` Value

Define the order request structure and create the order specifying the `timeOut` value of 200 milliseconds.

```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
createOrder(c,reqStruct,'timeOut',200)

r =

 EMSX_SEQUENCE: 354646
```

See Also

`createOrderAndRoute` | `orders` | `modifyOrder` | `deleteOrder` | `routes` | `routeOrder`

Related Examples

- “Bloomberg EMSX Order Management” on page 3-14
- “Bloomberg EMSX Route Management” on page 3-19
- “Bloomberg EMSX Order and Route Management” on page 3-24

Concepts

- “Workflow for Bloomberg EMSX” on page 2-2

Purpose

Create and route Bloomberg EMSX order

Syntax

```
R = createOrderAndRoute(c,reqStruct)
R = createOrderAndRoute(c,reqStruct,Name,Value)
```

Description

`R = createOrderAndRoute(c,reqStruct)` creates and routes a Bloomberg EMSX order and returns the order sequence number, route ID, and status message using the default event handler.

`R = createOrderAndRoute(c,reqStruct,Name,Value)` uses additional options specified by one or more `Name`, `Value` pair arguments. Create and route a Bloomberg EMSX order with optional name-value arguments to specify a custom event handler or timeout value for the event handler.

Note Name-value pair arguments can be input as a single input structure containing some or all of the property fields, for example:

```
p.timeOut = 1000;
p.useDefaultEventHandler = false;
createOrderAndRoute(c,reqStruct,p)
c.processEvent
```

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

reqStruct - Order request structure

structure

Order request structure, specified using EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `reqStruct`.

Example:
```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
```

Data Types
`struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
createOrderAndRoute(c,reqStruct,'useDefaultEventHandler',false)
```

'useDefaultEventHandler' - Flag for event handler preference

`true` (default) | logical with value `true` or `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, as specified by the string `true` or `false`. When this flag is set to the default, `true`, the default event handler is used. If a custom event handler is used, this flag must be set to `false`.

Example: `'useDefaultEventHandler',false`

Data Types
`logical`

'timeOut' - Connection timeout value for event handler for Bloomberg EMSX service

500 milliseconds (default) | nonnegative integer

Connection timeout value, specified as a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types
char

Output Arguments

R - Return status for order event

structure

Return status for the order event, returned as a structure.

Examples

Create and Route Bloomberg EMSX Order Using Default Event Handler

Define the order request structure and create and then route the order.

```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
r = createOrderAndRoute(c,reqStruct)
```

```
  r =

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

Create and Route Bloomberg EMSX Order Using Custom Event Handler

Define the order request structure and create and then route the order.

```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
```

```
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
createOrderAndRoute(c,reqStruct,'useDefaultEventHandler',false)
processEvent(c)

 CreateOrderAndRoute = {

     EMSX_SEQUENCE = 335877

     EMSX_ROUTE_ID = 1

     MESSAGE = Order created and routed

     }
```

Create and Route Bloomberg EMSX Order Using `timeOut` Value

Define the order request structure. Then create and route the order and assign a `timeOut` value of 200 milliseconds.

```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
createOrderAndRoute(c,reqStruct,'timeOut',200)

 r =

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

See Also

`createOrder` | `createOrderAndRouteWithStrat` | `orders` | `deleteOrder` | `routes` | `routeOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

createOrderAndRouteWithStrat

Purpose

Create and route Bloomberg EMSX order with strategies

Syntax

```
R = createOrderAndRouteWithStrat(c,reqStruct,stratStruct)
R =
createOrderAndRouteWithStrat(c,reqStruct,stratStruct,Name,Value)
```

Description

`R = createOrderAndRouteWithStrat(c,reqStruct,stratStruct)` creates and routes a Bloomberg EMSX order with strategies and returns the order sequence number, route ID, and status message using the default event handler.

`R = createOrderAndRouteWithStrat(c,reqStruct,stratStruct,Name,Value)` uses additional options specified by one or more `Name`, `Value` pair arguments. Create and route a Bloomberg EMSX order with strategies using optional name-value arguments to specify a custom event handler or timeout value for the event handler.

Note Name-value pair arguments can be input as a single input structure containing some or all of the property fields, for example:

```
p.timeOut = 1000;
createOrderAndRouteWithStrat(c,reqStruct,stratStruct,p)
```

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

reqStruct - Order request structure

structure

Order request structure, specified using EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `reqStruct`.

Example:
```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
```

Data Types
`struct`

`stratStruct` - Order strategies structure
structure

Order strategies structure, specified by the elements of the fields `EMSX_STRATEGY_FIELD_INDICATORS` and `EMSX_STRATEGY_FIELDS` in the `stratStruct`. In addition, the field elements of `stratStruct` must align with the fields for the strategy specified by `STRATSTRUCT.EMSX_STRATEGY_NAME`. For more information on strategy fields and ordering, see `getBrokerInfo`.

When using `stratStruct`, set `STRATSTRUCT.EMSX_STRATEGY_FIELD_INDICATORS` equal to `0` for each field so that the field data setting in `STRATSTRUCT.EMSX_FIELD_DATA` is used. Also set `STRATSTRUCT.EMSX_STRATEGY_FIELD_INDICATORS` equal to `1` to ignore the data in `STRATSTRUCT.EMSX_FIELD_DATA`.

Example:
```
stratStruct.EMSX_STRATEGY_NAME = 'SSP';
stratStruct.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0
0]);
stratStruct.EMSX_STRATEGY_FIELDS =
{'09:30:00','14:30:00',50};
```

Data Types
`struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `r = createOrderAndRouteWithStrat(c,reqStruct,stratStruct,'useDefaultEventHanc`

'useDefaultEventHandler' - Flag for event handler preference

`true` (default) | logical with value `true` or `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, as specified by the string `true` or `false`. When this flag is set to the default, `true`, the default event handler is used. If a custom event handler is used, this flag must be set to `false`.

Example: `'useDefaultEventHandler',false`

Data Types
`logical`

'timeOut' - Connection timeout value for event handler for Bloomberg EMSX service

500 milliseconds (default) | nonnegative integer

Connection timeout value, specified as a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types
`char`

Output Arguments

R - Return status for order event

structure

Return status for the order event returned as a structure.

Examples

Create and Route Bloomberg EMSX Order with Strategies Using Default Event Handler

Define the order request structure and strategies structure and then create and route the order.

```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
stratStruct.EMSX_STRATEGY_NAME = 'SSP';
stratStruct.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
stratStruct.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
r = createOrderAndRouteWithStrat(c,reqStruct,stratStruct)
```

```
 r =

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

Create and Route Bloomberg EMSX Order with Strategies Using Custom Event Handler

Define the order request structure and strategies structure and then create and route the order.

```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
stratStruct.EMSX_STRATEGY_NAME = 'SSP';
```

```
stratStruct.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
stratStruct.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
r = createOrderAndRouteWithStrat(c,reqStruct,stratStruct,'useDefaultEventHandler',false)
processEvent(c)
```

```
CreateOrderAndRouteWithStrat = {

     EMSX_SEQUENCE = 335877

     EMSX_ROUTE_ID = 1

     MESSAGE = Order created and routed

     }
```

Create and Route Bloomberg EMSX Order with Strategies Using `timeOut` Value

Define the order request structure and then create and route the order and assign a `timeOut` value of 200 milliseconds.

```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
stratStruct.EMSX_STRATEGY_NAME = 'SSP';
stratStruct.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
stratStruct.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
r = createOrderAndRouteWithStrat(c,reqStruct,stratStruct,...
            'timeOut',200)
```

```
 r =

        EMSX_SEQUENCE: 335877
        EMSX_ROUTE_ID: 1
```

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

See Also

`getBrokerInfo` | `createOrder` | `orders` | `deleteOrder` | `routes` | `routeOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

deleteOrder

Purpose

Delete Bloomberg EMSX order

Syntax

```
R = deleteOrder(c,reqStruct)
R = deleteOrder(c,reqStruct,Name,Value)
```

Description

`R = deleteOrder(c,reqStruct)` deletes a Bloomberg EMSX order and returns a status message using the default event handler.

`R = deleteOrder(c,reqStruct,Name,Value)` uses additional options specified by one or more `Name`, `Value` pair arguments. Delete a Bloomberg EMSX order using optional name-value arguments to specify a custom event handler or timeout value for the event handler.

Note Name-value pair arguments can be input as a single input structure containing some or all of the property fields, for example:

```
p.timeOut = 1000;
deleteOrder(c,reqStruct,p)
```

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

reqStruct - Order request structure

structure | integer for `EMSX_SEQUENCE` number

Order request structure, specified as a `reqStruct` or `EMSX_SEQUENCE` number.

Example:

```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_TIF = 'DAY';
```

```
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
```

Data Types
`int32 | struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
deleteOrder(c,int32(335877),'useDefaultEventHandler',false)
```

'useDefaultEventHandler' - Flag for event handler preference

`true` (default) | logical with value `true` or `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, as specified by the string `true` or `false`. When this flag is set to the default, `true`, the default event handler is used. If a custom event handler is used, this flag must be set to `false`.

Example: `'useDefaultEventHandler',false`

Data Types
`logical`

'timeOut' - Connection timeout value for event handler for Bloomberg EMSX service

500 milliseconds (default) | nonnegative integer

Connection timeout value, specified as a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types
`char`

Output Arguments

R - Return status for requested event
structure

Return status for the order event, returned as a structure.

Examples

Delete Bloomberg EMSX Order Using Default Event Handler

Define the EMSX_SEQUENCE for the order and then delete the order.

```
reqStruct.EMSX_SEQUENCE = int32(335877)
r = deleteOrder(c,reqStruct)
```

```
 r =

      STATUS: 'O'
     MESSAGE: 'Order deleted'
```

Delete Bloomberg EMSX Order Using Custom Event Handler

Define the EMSX_SEQUENCE for the order and then delete the order.

```
reqStruct.EMSX_SEQUENCE = int32(335877)
deleteOrder(c,int32(335877),'useDefaultEventHandler',false)
processEvent(c)
```

```
DeleteOrder = {

        STATUS = O

        MESSAGE = Order deleted

    }
```

Delete Bloomberg EMSX Order Using `timeOut` Value

Define the EMSX_SEQUENCE for the order and then delete the order.

```
reqStruct.EMSX_SEQUENCE = int32(335877)
deleteOrder(c,int32(335877),'timeOut',200)
```

```
r =

     STATUS: 'O'
```

See Also

`createOrderAndRoute` | `orders` | `createOrder` | `routes` | `modifyOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

deleteRoute

Purpose

Delete Bloomberg EMSX route

Syntax

```
R = deleteRoute(c,reqStruct)
R = deleteRoute(c,reqStruct,Name,Value)
```

Description

`R = deleteRoute(c,reqStruct)` deletes a Bloomberg EMSX route and returns a status message using the default event handler.

`R = deleteRoute(c,reqStruct,Name,Value)` uses additional options specified by one or more `Name`, `Value` pair arguments. Delete a Bloomberg EMSX route using optional name-value arguments to specify a custom event handler or timeout value for the event handler.

Note Name-value pair arguments can be input as a single input structure containing some or all of the property fields, for example:

```
p.timeOut = 1000;
deleteRoute(c,reqStruct,p)
```

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

reqStruct - Order request structure

structure | integer for `EMSX_SEQUENCE` number

Order request structure, specified as a `reqStruct` or `EMSX_SEQUENCE` number.

Example:

```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_TIF = 'DAY';
```

```
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
```

Data Types
`int32 | struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`deleteRoute(c,reqStruct,'useDefaultEventHandler',false)`

'useDefaultEventHandler' - Flag for event handler preference

`true` (default) | logical with value `true` or `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, as specified by the string `true` or `false`. When this flag is set to the default, `true`, the default event handler is used. If a custom event handler is used, this flag must be set to `false`.

Example: `'useDefaultEventHandler',false`

Data Types
`logical`

'timeOut' - Connection timeout value for event handler for Bloomberg EMSX service

500 milliseconds (default) | nonnegative integer

Connection timeout value, specified as a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types
`char`

Output Arguments

R - Return status for requested event

structure

Return status for the order event returned as a structure.

Examples

Delete Route for Bloomberg EMSX Order Using Default Event Handler

Define the `reqStruct` values for `EMSX_SEQUENCE` and `EMSX_ROUTE_ID`. Then delete the route.

```
reqStruct.EMSX_SEQUENCE = int32(335877)
reqStruct.EMSX_ROUTE_ID = int32(1)
r = deleteRoute(c,reqStruct)
```

```
 r =

      STATUS: '0'
     MESSAGE: 'Route deleted'
```

Delete Route for Bloomberg EMSX Order Using Custom Event Handler

Define the `reqStruct` values for `EMSX_SEQUENCE` and `EMSX_ROUTE_ID`. Then delete the route.

```
reqStruct.EMSX_SEQUENCE = int32(335877)
reqStruct.EMSX_ROUTE_ID = int32(1)
deleteRoute(c,reqStruct,'useDefaultEventHandler',false)
processEvent(c)
```

```
DeleteRoute = {

        STATUS = 0

        MESSAGE = Route deleted

    }
```

Delete Route for Bloomberg EMSX Order Using `timeOut` Value

Define the `reqStruct` values for `EMSX_SEQUENCE` and `EMSX_ROUTE_ID`. Then delete the route.

```
reqStruct.EMSX_SEQUENCE = int32(335877)
reqStruct.EMSX_ROUTE_ID = int32(1)
deleteRoute(c,int32(335877),'timeOut',200)
```

```
 r =

      STATUS: 'O'
     MESSAGE: 'Route deleted'
```

See Also

`createOrderAndRoute` | `orders` | `createOrder` | `routes` | `modifyRoute`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

getAllFieldMetaData

Purpose

Obtain Bloomberg EMSX field information

Syntax

```
R = getAllFieldMetaData(c)
```

Description

`R = getAllFieldMetaData(c)` returns the Bloomberg EMSX field information given the connection handle `c`.

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

Output Arguments

R - Return information for all fields

structure

Return information, returned as a structure for all fields supported by Bloomberg EMSX service. This information is used to create a request structure (`reqStruct`) for orders.

Examples

Request All Field Information for EMSX

Request all fields supported by Bloomberg EMSX service.

```
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]
```

where

```
{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]
```

See Also

`emsx`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

getBrokerInfo

Purpose

Obtain Bloomberg EMSX broker and strategy information

Syntax

```
R = getBrokerInfo(c,reqStruct)
```

Description

`R = getBrokerInfo(c,reqStruct)` obtains Bloomberg EMSX broker and strategy information and returns a status message using the default event handler.

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

reqStruct - Order request structure

structure

Order request structure, specified using EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `reqStruct`.

Example: `reqStruct.EMSX_TICKER = 'ABCD US Equity';`

Data Types
`struct`

Output Arguments

R - Return status for requested event

structure

Return status for the order event, returned as a structure.

Examples

Obtain Broker Information for Bloomberg EMSX

Define the `reqstruct` for one item and then request broker information.

```
reqStruct.EMSX_TICKER = 'ABCD US Equity';
r = getBrokerInfo(c,reqStruct)
```

```
 r =
```

```
        EMSX_BROKERS: {2x1 cell}
```

Define the `reqstruct` for two items and then request broker information.

```
reqStruct.EMSX_TICKER = 'ABCD US Equity';
reqStruct.EMSX_BROKER = 'BMTB';
r = getBrokerInfo(c,reqStruct)
```

```
 r =

        EMSX_STRATEGIES: {16x1 cell}
```

Define the `reqstruct` for three items then request broker information.

```
reqStruct.EMSX_TICKER = 'ABCD US Equity';
reqStruct.EMSX_BROKER = 'BMTB';
reqStruct.EMSX_STRATEGY = 'SSP';
r = getBrokerInfo(c,reqStruct)
```

```
r =

        FieldName: {3x1 cell}
        Disable: {3x1 cell}
        StringValue: {3x1 cell}
```

See Also

`getRouteInfo` | `getOrderInfo` | `createOrder` | `createOrderAndRoute` | `orders` | `modifyOrder` | `routes` | `deleteOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

getOrderInfo

Purpose

Obtain Bloomberg EMSX order information

Syntax

```
R = getOrderInfo(c,reqStruct)
R = getOrderInfo(c,reqStruct,Name,Value)
```

Description

`R = getOrderInfo(c,reqStruct)` obtains Bloomberg EMSX order information and returns a status message using the default event handler.

`R = getOrderInfo(c,reqStruct,Name,Value)` uses additional options specified by one or more `Name`, `Value` pair arguments. Obtain Bloomberg EMSX order information using optional name-value arguments to specify a custom event handler or timeout value for the event handler.

Note Name-value pair arguments can be input as a single input structure containing some or all of the property fields, for example:

```
p.timeOut = 1000;
getOrderInfo(c,reqStruct,p)
```

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

reqStruct - Order request structure

structure | integer for `EMSX_SEQUENCE` number

Order request structure, specified using EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `reqStruct`.

Note `EMSX_SEQUENCE` must denote an existing order sequence number.

Example: `reqStruct.EMSX_SEQUENCE = int32(335877);`

Data Types
`int32 | struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `r = getOrderInfo(c,reqStruct,'useDefaultEventHandler',false)`

'useDefaultEventHandler' - Flag for event handler preference

`true` (default) | logical with value `true` or `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, as specified by the string `true` or `false`. When this flag is set to the default, `true`, the default event handler is used. If a custom event handler is used, this flag must be set to `false`.

Example: `'useDefaultEventHandler',false`

Data Types
`logical`

'timeOut' - Connection timeout value for event handler for Bloomberg EMSX service

500 milliseconds (default) | nonnegative integer

Connection timeout value, specified as a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types
`char`

Output Arguments

R - Return status for requested event

structure

Return status for the order event, returned as a structure.

Examples

Obtain Order Information for Bloomberg EMSX Using Default Event Handler

Define the `reqstruct` and note that `EMSX_SEQUENCE` must denote an existing order.

```
reqStruct.EMSX_SEQUENCE = int32(335877);
r = getOrderInfo(c,reqStruct)
```

```
r =

              EMSX_TICKER: 'IBM'
            EMSX_EXCHANGE: 'US'
                EMSX_SIDE: 'BUY'
            EMSX_POSITION: 'BUY'
            EMSX_PORT_MGR: 'CF'
              EMSX_TRADER: 'CF'
               EMSX_NOTES: ''
              EMSX_AMOUNT: 400
         EMSX_IDLE_AMOUNT: 150
             EMSX_WORKING: 0
              EMSX_FILLED: 250
          EMSX_TS_ORDNUM: 250
         EMSX_LIMIT_PRICE: 0
           EMSX_AVG_PRICE: 189.7900
                EMSX_FLAG: 2
            EMSX_SUB_FLAG: 0
         EMSX_YELLOW_KEY: 'Equity'
```

```
      EMSX_BASKET_NAME: ''
EMSX_ORDER_CREATE_DATE: '12/06/12'
EMSX_ORDER_CREATE_TIME: '14:28:37'
       EMSX_ORDER_TYPE: 'MKT'
              EMSX_TIF: 'DAY'
           EMSX_BROKER: 'BB'
      EMSX_TRADER_UUID: '1244972'
  EMSX_STEP_OUT_BROKER: ''
```

Obtain Order Information for Bloomberg EMSX Using Custom Event Handler

Define the `reqstruct` and note that `EMSX_SEQUENCE` and must denote an existing order.

```
reqStruct.EMSX_SEQUENCE = int32(335877);
r = getOrderInfo(c,reqStruct,'useDefaultEventHandler',false)

processEvent(c)

OrderRouteFields = {

    MSG_TYPE = E

    EVENT_STATUS = 1

    API_SEQ_NUM = 8

    EMSX_SEQUENCE = 0

    EMSX_AMOUNT = 0

    EMSX_FILLED = 0

    EMSX_AVG_PRICE = 0.0

    EMSX_WORKING = 0
```

```
    EMSX_TIME_STAMP = 0

    ...

}

OrderInfo = {

    EMSX_TICKER = IBM

    EMSX_EXCHANGE = US

    EMSX_SIDE = BUY

    EMSX_POSITION = BUY

    EMSX_PORT_MGR = CG

    EMSX_TRADER = CG

    EMSX_NOTES =

    EMSX_AMOUNT = 400

    EMSX_IDLE_AMOUNT = 150

    EMSX_WORKING = 0

    EMSX_FILLED = 250

    EMSX_TS_ORDNUM = -381490

    EMSX_LIMIT_PRICE = 0.0

    EMSX_AVG_PRICE = 189.7899963378906
```

```
    EMSX_FLAG = 2

    EMSX_SUB_FLAG = 0

    EMSX_YELLOW_KEY = Equity

    EMSX_BASKET_NAME =

    EMSX_ORDER_CREATE_DATE = 12/06/12

    EMSX_ORDER_CREATE_TIME = 14:28:37

    EMSX_ORDER_TYPE = MKT

    EMSX_TIF = DAY

    EMSX_BROKER = BB

    EMSX_TRADER_UUID = 1244972

    EMSX_STEP_OUT_BROKER =

}
```

Obtain Order Information for Bloomberg EMSX Using `timeOut` Value

Define the `reqstruct` and note that `EMSX_SEQUENCE` must denote an existing order.

```
reqStruct.EMSX_SEQUENCE = int32(335877);
r = getOrderInfo(c,reqStruct,'timeOut',200)
```

```
r =

              EMSX_TICKER: 'IBM'
            EMSX_EXCHANGE: 'US'
                EMSX_SIDE: 'BUY'
```

```
          EMSX_POSITION: 'BUY'
          EMSX_PORT_MGR: 'CF'
            EMSX_TRADER: 'CF'
             EMSX_NOTES: ''
            EMSX_AMOUNT: 400
       EMSX_IDLE_AMOUNT: 150
           EMSX_WORKING: 0
            EMSX_FILLED: 250
         EMSX_TS_ORDNUM: 250
       EMSX_LIMIT_PRICE: 0
         EMSX_AVG_PRICE: 189.7900
              EMSX_FLAG: 2
          EMSX_SUB_FLAG: 0
        EMSX_YELLOW_KEY: 'Equity'
       EMSX_BASKET_NAME: ''
EMSX_ORDER_CREATE_DATE: '12/06/12'
EMSX_ORDER_CREATE_TIME: '14:28:37'
        EMSX_ORDER_TYPE: 'MKT'
               EMSX_TIF: 'DAY'
            EMSX_BROKER: 'BB'
      EMSX_TRADER_UUID: '1244972'
  EMSX_STEP_OUT_BROKER: ''
```

See Also

`getRouteInfo` | `getBrokerInfo` | `createOrder` | `createOrderAndRoute` | `orders` | `modifyOrder` | `routes` | `deleteOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

Purpose

Obtain Bloomberg EMSX route information

Syntax

```
R = getRouteInfo(c,reqStruct)
R = getRouteInfo(c,reqStruct,Name,Value)
```

Description

`R = getRouteInfo(c,reqStruct)` obtains Bloomberg EMSX route information and returns a status message using the default event handler.

`R = getRouteInfo(c,reqStruct,Name,Value)` uses additional options specified by one or more `Name`, `Value` pair arguments. Obtain Bloomberg EMSX route information using optional name-value arguments to specify a custom event handler or timeout value for the event handler.

Note Name-value pair arguments can be input as a single input structure containing some or all of the property fields, for example:

```
p.timeOut = 1000;
getRouteInfo(c,reqStruct,p)
```

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

reqStruct - Order request structure

structure | integer for `EMSX_SEQUENCE` and `EMSX_ROUTE_ID`

Order request structure, specified using EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `reqStruct`.

Note `EMSX_SEQUENCE` must denote an existing order sequence number and `EMSX_ROUTE_ID` must denote an existing route ID.

Example:
```
reqStruct.EMSX_SEQUENCE = int32(335877);
reqStruct.EMSX_ROUTE_ID = int32(1);
```

Data Types
`int32 | struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
r = getRouteInfo(c,reqStruct,'useDefaultEventHandler',false)
```

'useDefaultEventHandler' - Flag for event handler preference

`true` (default) | logical with value `true` or `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, as specified by the string `true` or `false`. When this flag is set to the default, `true`, the default event handler is used. If a custom event handler is used, this flag must be set to `false`.

Example: `'useDefaultEventHandler',false`

Data Types
`logical`

'timeOut' - Connection timeout value for event handler for Bloomberg EMSX service

500 milliseconds (default) | nonnegative integer

Connection timeout value, specified as a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types
`char`

Output Arguments

R - Return status for requested event

structure

Return status for the order event, returned as a structure.

Examples

Obtain Route Information for Bloomberg EMSX Using Default Event Handler

Define the `reqstruct` and note that `EMSX_SEQUENCE` and `EMSX_ROUTE_ID` must denote an existing order.

```
reqStruct.EMSX_SEQUENCE = int32(335877);
reqStruct.EMSX_ROUTE_ID = int32(1);
r = getRouteInfo(c,reqStruct)
```

```
r =

                  EMSX_AVG_PRICE: 189.7900
                      EMSX_YIELD: 0
         EMSX_ROUTE_CREATE_DATE: 20121206
         EMSX_ROUTE_CREATE_TIME: 142837
    EMSX_ROUTE_LAST_UPDATE_DATE: 20121206
    EMSX_ROUTE_LAST_UPDATE_TIME: 143251
               EMSX_SETTLE_DATE: 20121211
                    EMSX_AMOUNT: 250
                    EMSX_FILLED: 250
           EMSX_IS_MANUAL_ROUTE: 0
                    EMSX_BROKER: 'BB'
                   EMSX_ACCOUNT: ''
                 EMSX_STATUS_ID: 199032
```

```
              EMSX_STATUS: 'Filled'
    EMSX_HAND_INSTRUCTION: 'ANY'
          EMSX_ORDER_TYPE: 'MKT'
                 EMSX_TIF: 'DAY'
              EMSX_LOC_ID: ''
          EMSX_LOC_BROKER: 'DAY'
          EMSX_STOP_PRICE: 0
        EMSX_BLOT_SEQ_NUM: 1
           EMSX_BLOT_DATE: 20121206
           EMSX_COMM_TYPE: 'DAY'
           EMSX_COMM_RATE: 0
    EMSX_USER_COMM_AMOUNT: 0
           EMSX_LSTTR2ID0: 1.3548e+09
           EMSX_LSTTR2ID1: 284950536
         EMSX_LIMIT_PRICE: 0
```

Obtain Route Information for Bloomberg EMSX Using Custom Event Handler

Define the `reqstruct` and note that `EMSX_SEQUENCE` and `EMSX_ROUTE_ID` must denote an existing order.

```
reqStruct.EMSX_SEQUENCE = int32(335877);
reqStruct.EMSX_ROUTE_ID = int32(1);
r = getRouteInfo(c,reqStruct,'useDefaultEventHandler',false)

processEvent(c)

RouteInfo = {

    EMSX_LIMIT_PRICE = 0.0

    EMSX_YIELD = 0.0

    EMSX_AVG_PRICE = 193.9600067138672

    EMSX_ROUTE_CREATE_DATE = 20121211
```

```
EMSX_ROUTE_CREATE_TIME = 101324

EMSX_ROUTE_LAST_UPDATE_DATE = 20121211

EMSX_ROUTE_LAST_UPDATE_TIME = 101325

EMSX_SETTLE_DATE = 20121214

EMSX_AMOUNT = 100

EMSX_FILLED = 50

EMSX_IS_MANUAL_ROUTE = 0

EMSX_BROKER = BB

EMSX_ACCOUNT =

EMSX_STATUS_ID = 51088

EMSX_STATUS = PtlFil

EMSX_HAND_INSTRUCTION = ANY

EMSX_ORDER_TYPE = MKT

EMSX_TIF = DAY

EMSX_LOC_ID =

EMSX_LOC_BROKER =

EMSX_STOP_PRICE = 0.0

EMSX_BLOT_SEQ_NUM = 2
```

```
    EMSX_BLOT_DATE = 20121211

    EMSX_COMM_TYPE =

    EMSX_COMM_RATE = 0.0

    EMSX_USER_COMM_AMOUNT = 0.0

    EMSX_LSTTR2ID0 = 1355238804

    EMSX_LSTTR2ID1 = 284950539

}
```

Obtain Route Information for Bloomberg EMSX Using `timeOut` Value

Define the `reqstruct` and note that `EMSX_SEQUENCE` and `EMSX_ROUTE_ID` must denote an existing order.

```
reqStruct.EMSX_SEQUENCE = int32(335877);
reqStruct.EMSX_ROUTE_ID = int32(1);
r = getRouteInfo(c,reqStruct,'timeOut',200)

r =

                EMSX_AVG_PRICE: 189.7900
                    EMSX_YIELD: 0
         EMSX_ROUTE_CREATE_DATE: 20121206
         EMSX_ROUTE_CREATE_TIME: 142837
    EMSX_ROUTE_LAST_UPDATE_DATE: 20121206
    EMSX_ROUTE_LAST_UPDATE_TIME: 143251
               EMSX_SETTLE_DATE: 20121211
                    EMSX_AMOUNT: 250
                    EMSX_FILLED: 250
          EMSX_IS_MANUAL_ROUTE: 0
                    EMSX_BROKER: 'BB'
                   EMSX_ACCOUNT: ''
```

```
          EMSX_STATUS_ID: 199032
             EMSX_STATUS: 'Filled'
EMSX_HAND_INSTRUCTION: 'ANY'
       EMSX_ORDER_TYPE: 'MKT'
              EMSX_TIF: 'DAY'
             EMSX_LOC_ID: ''
         EMSX_LOC_BROKER: 'DAY'
         EMSX_STOP_PRICE: 0
      EMSX_BLOT_SEQ_NUM: 1
         EMSX_BLOT_DATE: 20121206
         EMSX_COMM_TYPE: 'DAY'
         EMSX_COMM_RATE: 0
EMSX_USER_COMM_AMOUNT: 0
         EMSX_LSTTR2ID0: 1.3548e+09
         EMSX_LSTTR2ID1: 284950536
       EMSX_LIMIT_PRICE: 0
```

See Also

`getOrderInfo` | `getBrokerInfo` | `createOrder` | `createOrderAndRoute` | `orders` | `modifyOrder` | `routes` | `deleteOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

modifyOrder

Purpose

Modify Bloomberg EMSX order

Syntax

```
R = modifyOrder(c,reqStruct)
R = modifyOrder(c,reqStruct,Name,Value)
```

Description

`R = modifyOrder(c,reqStruct)` modifies a Bloomberg EMSX order and returns a status message using the default event handler.

`R = modifyOrder(c,reqStruct,Name,Value)` uses additional options specified by one or more `Name`, `Value` pair arguments. Modify a Bloomberg EMSX order using optional name-value arguments to specify a custom event handler or timeout value for the event handler.

Note Name-value pair arguments can be input as a single input structure containing some or all of the property fields, for example:

```
modifyOrder(c,reqStruct,'useDefaultEventHandler',false)
processEvent(c)
```

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

reqStruct - Order request structure

structure

Order request structure, specified using EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `reqStruct`.

Example:
```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_TIF = 'DAY';
```

```
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
```

Data Types
`struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
modifyOrder(c,reqStruct,'useDefaultEventHandler',false)
```

'useDefaultEventHandler' - Flag for event handler preference

`true` (default) | logical with value `true` or `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, as specified by the string `true` or `false`. When this flag is set to the default, `true`, the default event handler is used. If a custom event handler is used, this flag must be set to `false`.

Example: `'useDefaultEventHandler',false`

Data Types
`logical`

'timeOut' - Connection timeout value for event handler for Bloomberg EMSX service

500 milliseconds (default) | nonnegative integer

Connection timeout value, specified as a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types
`char`

Output Arguments

R - Return status for requested event

structure

Return status for the order event, returned as a structure.

Examples

Modify Order for Bloomberg EMSX Using Default Event Handler

Define the `reqStruct` and then modify the order.

```
reqStruct.EMSX_SEQUENCE = int32(335877)
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(200);
r = modifyOrder(c,reqStruct)
```

```
r =

     EMSX_SEQUENCE: 3335877
           MESSAGE: 'Order Modified'
```

Modify Order for Bloomberg EMSX Using Custom Event Handler

Define the `reqStruct` and then modify the order.

```
reqStruct.EMSX_SEQUENCE = int32(335877)
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(200);
modifyOrder(c,reqStruct,'useDefaultEventHandler',false)
processEvent(c)
```

```
  ModifyOrder = {

     EMSX_SEQUENCE = 335877

     MESSAGE = Order Modified

     }
```

Modify Order for Bloomberg Using `timeOut` Value

Define the `reqStruct` and then modify the order.

```
reqStruct.EMSX_SEQUENCE = int32(335877)
reqStruct.EMSX_ROUTE_ID = int32(1)
modifyOrder(c,int32(335877),'timeOut',200)
```

```
r =

     EMSX_SEQUENCE: 3335877
           MESSAGE: 'Order Modified'
```

See Also

`createOrderAndRoute` | `orders` | `createOrder` | `routes` | `deleteOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

modifyRoute

Purpose

Modify Bloomberg EMSX route

Syntax

```
R = modifyRoute(c,reqStruct)
R = modifyRoute(c,reqStruct,Name,Value)
```

Description

`R = modifyRoute(c,reqStruct)` modifies a Bloomberg EMSX route and returns a status message using the default event handler.

`R = modifyRoute(c,reqStruct,Name,Value)` uses additional options specified by one or more `Name`, `Value` pair arguments. Modify a Bloomberg EMSX route using optional name-value arguments to specify a custom event handler or timeout value for the event handler.

Note Name-value pair arguments can be input as a single input structure containing some or all of the property fields, for example:

```
p.timeOut = 1000;
modifyRoute(c,reqStruct,p)
```

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

reqStruct - Order request structure

structure

Order request structure, specified using EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `reqStruct`.

Example:
```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_TIF = 'DAY';
```

```
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
```

Data Types
`struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`modifyRoute(c,reqStruct,'useDefaultEventHandler',false)`

'useDefaultEventHandler' - Flag for event handler preference

`true` (default) | logical with value `true` or `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, as specified by the string `true` or `false`. When this flag is set to the default, `true`, the default event handler is used. If a custom event handler is used, this flag must be set to `false`.

Example: `'useDefaultEventHandler',false`

Data Types
`logical`

'timeOut' - Connection timeout value for event handler for Bloomberg EMSX service

500 milliseconds (default) | nonnegative integer

Connection timeout value, specified as a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types
`char`

Output Arguments

R - Return status
structure

Return status for the order event, returned as a structure.

Examples

Modify Route for Bloomberg EMSX Order Using Default Event Handler

Define the `reqStruct` and then modify the route.

```
reqStruct.EMSX_SEQUENCE = int32(335877)
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(200);
r = modifyRoute(c,reqStruct)
```

```
 r =

     EMSX_SEQUENCE: 3335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Modified'
```

Modify Route for Bloomberg EMSX Order Using Custom Event Handler

Define the `reqStruct` and then modify the route.

```
reqStruct.EMSX_SEQUENCE = int32(335877)
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(200);
modifyRoute(c,reqStruct,'useDefaultEventHandler',false)
processEvent(c)
```

```
ModifyRoute = {

     EMSX_SEQUENCE = 335877

     EMSX_ROUTE_ID = 1

     MESSAGE = Route Modified
```

```
    }
```

Modify Route for Bloomberg Using `timeOut` Value

Define the `reqStruct` and then modify the route.

```
reqStruct.EMSX_SEQUENCE = int32(335877)
reqStruct.EMSX_ROUTE_ID = int32(1)
modifyRoute(c,int32(335877),'timeOut',200)
```

```
 r =

    EMSX_SEQUENCE: 3335877
    EMSX_ROUTE_ID: 1
          MESSAGE: 'Order Modified'
```

See Also

`createOrderAndRoute` | `orders` | `createOrder` | `routes` | `deleteOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

modifyRouteWithStrat

Purpose

Modify route with strategies for Bloomberg EMSX

Syntax

```
R = modifyRouteWithStrat(c,reqStruct,stratStruct)
R =
modifyRouteWithStrat(c,reqStruct,stratStruct,Name,Value)
```

Description

`R = modifyRouteWithStrat(c,reqStruct,stratStruct)` modifies a Bloomberg EMSX route with strategies and returns the order sequence number, route ID, and status message using the default event handler.

`R =`
`modifyRouteWithStrat(c,reqStruct,stratStruct,Name,Value)` uses additional options specified by one or more `Name`, `Value` pair arguments. Modify a Bloomberg EMSX route with strategies using optional name-value arguments to specify a custom event handler or timeout value for the event handler.

Note Name-value pair arguments can be input as a single input structure containing some or all of the property fields, for example:

```
p.timeOut = 1000;
modifyRouteWithStrat(c,reqStruct,stratStruct,p)
```

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

reqStruct - Order request structure

structure

Order request structure, specified using EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `reqStruct`.

Example:
```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
```

Data Types
`struct`

`stratStruct` - Order strategies structure
structure

Order strategies structure specified by the elements of the fields `EMSX_STRATEGY_FIELD_INDICATORS` and `EMSX_STRATEGY_FIELDS` in the `stratStruct`. In addition, the field elements of `stratStruct` must align with the fields for the strategy specified by `STRATSTRUCT.EMSX_STRATEGY_NAME`. For more information on strategy fields and ordering, see `getBrokerInfo`.

When using `stratStruct`, set `STRATSTRUCT.EMSX_STRATEGY_FIELD_INDICATORS` equal to `0` for each field so that the field data setting in `STRATSTRUCT.EMSX_FIELD_DATA` is used. Also set `STRATSTRUCT.EMSX_STRATEGY_FIELD_INDICATORS` equal to `1` to ignore the data in `STRATSTRUCT.EMSX_FIELD_DATA`.

Example:
```
stratStruct.EMSX_STRATEGY_NAME = 'SSP';
stratStruct.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0
0]);
stratStruct.EMSX_STRATEGY_FIELDS =
{'09:30:00','14:30:00',50};
```

Data Types
`struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can

specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `r = modifyRouteWithStrat(c,reqStruct,stratStruct,'useDefaultEventHandler',fal`

'useDefaultEventHandler' - Flag for event handler preference

`true` (default) | logical with value `true` or `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, as specified by the string `true` or `false`. When this flag is set to the default, `true`, the default event handler is used. If a custom event handler is used, this flag must be set to `false`.

Example: `'useDefaultEventHandler',false`

Data Types
`logical`

'timeOut' - Connection timeout value for event handler for Bloomberg EMSX service

500 milliseconds (default) | nonnegative integer

Connection timeout value, specified as a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types
`char`

Output Arguments

R - Return status for order event

structure

Return status for the order event, returned as a structure.

Examples

Modify Bloomberg EMSX Route with Strategies Using Default Event Handler

Define the order request structure and strategies structure and then modify the route.

```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
stratStruct.EMSX_STRATEGY_NAME = 'SSP';
stratStruct.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
stratStruct.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
r = modifyRouteWithStrat(c,reqStruct,stratStruct)
```

```
 r =

        EMSX_SEQUENCE: 335877
        EMSX_ROUTE_ID: 1
              MESSAGE: 'Order Modified'
```

Modify Bloomberg EMSX Route with Strategies Using Custom Event Handler

Define the order request structure and strategies structure and then modify the route.

```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
stratStruct.EMSX_STRATEGY_NAME = 'SSP';
```

```
stratStruct.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
stratStruct.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
r = modifyRouteWithStrat(c,reqStruct,stratStruct,'useDefaultEventHandler',false)
processEvent(c)
```

```
ModifyRouteWithStrat = {

     EMSX_SEQUENCE = 335877

     EMSX_ROUTE_ID = 1

     MESSAGE = Route modified

     }
```

Modify Bloomberg EMSX Route with Strategies Using `timeOut` Value

Define the order request structure and modify route and assign a `timeOut` value of 200 milliseconds.

```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
stratStruct.EMSX_STRATEGY_NAME = 'SSP';
stratStruct.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
stratStruct.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
modifyRouteWithStrat(c,reqStruct,stratStruct,'timeOut',200)
```

```
 r =

        EMSX_SEQUENCE: 335877
        EMSX_ROUTE_ID: 1
              MESSAGE: 'Order Modified'
```

See Also

`getBrokerInfo` | `createOrderAndRouteWithStrat` | `createOrder` | `orders` | `deleteOrder` | `routes` | `routeOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

orders

Purpose

Obtain Bloomberg order subscription

Syntax

```
R = orders(c,fields)
R = orders(c,fields,Name,Value)
```

Description

`R = orders(c,fields)` subscribes to Bloomberg EMSX fields and returns information about outstanding orders using the default event handler.

`R = orders(c,fields,Name,Value)` uses additional options specified by one or more `Name`, `Value` pair arguments. Subscribe to Bloomberg EMSX fields and return information about outstanding orders using optional name-value arguments to specify a custom event handler or timeout value for the event handler.

Note Name-value pair arguments can be input as a single input structure containing some or all of the property fields, for example:

```
p.timeOut = 1000;
orders(c,{'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'},p)
```

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

fields - EMSX field information

cell array

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`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
orders(c,{'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'},'useDefaultEventH
```

'useDefaultEventHandler' - Flag for event handler preference

`true` (default) | logical with value `true` or `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, as specified by the string `true` or `false`. When this flag is set to the default, `true`, the default event handler is used. If a custom event handler is used, this flag must be set to `false`.

Example: `'useDefaultEventHandler',false`

Data Types
`logical`

'timeOut' - Connection timeout value for event handler for Bloomberg EMSX service

`500` milliseconds (default) | nonnegative integer

Connection timeout value, specified as a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types
`char`

Output Arguments

R - Return status

structure

Return status for order subscription information for existing orders, returned as a structure.

Examples

Request Order Subscription for Bloomberg EMSX Orders Using Default Event Handler

Request order subscription for existing EMSX orders.

```
orders(c,{'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'})
```

```
orders =

                    MSG_TYPE: {7x1 cell}
                MSG_SUB_TYPE: {7x1 cell}
                EVENT_STATUS: [7x1 int32]
                  API_SEQ_NUM: [7x1 int64]
               EMSX_SEQUENCE: [7x1 int32]
               EMSX_ROUTE_ID: [7x1 int32]
                EMSX_FILL_ID: [7x1 int32]
                   EMSX_SIDE: {7x1 cell}
                 EMSX_AMOUNT: [7x1 int32]
                 EMSX_FILLED: [7x1 int32]
              EMSX_AVG_PRICE: [7x1 double]
                 EMSX_BROKER: {7x1 cell}
                EMSX_WORKING: [7x1 int32]
                 EMSX_TICKER: {7x1 cell}
               EMSX_EXCHANGE: {7x1 cell}
     EMSX_ROUTE_CREATE_TIME: {7x1 cell}
                   EMSX_TIF: {7x1 cell}
EMSX_ROUTE_LAST_UPDATE_TIME: {7x1 cell}

                    ....
```

Request Order Subscription for Bloomberg EMSX Orders Using Custom Event Handler

Use the custom event handler.

```
orders(c,{'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'},'useDefaultEventHandler',false)
processEvent(c)
```

```
OrderRouteFields = {

    MSG_TYPE = E

    EVENT_STATUS = 1

    API_SEQ_NUM = 2

    EMSX_SEQUENCE = 0

    EMSX_AMOUNT = 0

    EMSX_FILLED = 0

    EMSX_AVG_PRICE = 0.0

    EMSX_WORKING = 0

    EMSX_TIME_STAMP = 0

    EMSX_ROUTE_PRICE = 0.0

    EMSX_LIMIT_PRICE = 0.0

    ...
```

Request Order Subscription for Bloomberg EMSX Orders Using `timeOut` Value

Use the `timeOut` value.

```
orders(c,{'EMSX_BROKER','EMSX_AMOUNT','EMSX_FILLED'},'timeOut',200)
```

```
 orders =

                        MSG_TYPE: {7x1 cell}
                    MSG_SUB_TYPE: {7x1 cell}
                    EVENT_STATUS: [7x1 int32]
                      API_SEQ_NUM: [7x1 int64]
                   EMSX_SEQUENCE: [7x1 int32]
                   EMSX_ROUTE_ID: [7x1 int32]
                    EMSX_FILL_ID: [7x1 int32]
                       EMSX_SIDE: {7x1 cell}
                     EMSX_AMOUNT: [7x1 int32]
                     EMSX_FILLED: [7x1 int32]
                  EMSX_AVG_PRICE: [7x1 double]
                     EMSX_BROKER: {7x1 cell}
                    EMSX_WORKING: [7x1 int32]
                     EMSX_TICKER: {7x1 cell}
                   EMSX_EXCHANGE: {7x1 cell}
         EMSX_ROUTE_CREATE_TIME: {7x1 cell}
                       EMSX_TIF: {7x1 cell}
    EMSX_ROUTE_LAST_UPDATE_TIME: {7x1 cell}

                                ...
```

See Also

`emsx` | `createOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

Purpose

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 object, `T` is the timer object associated with the event handler, and `Subs` is the Bloomberg order subscription.

Input Arguments

`c` - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

Output Arguments

`T` - Timer for event handler

string

Timer for the event handler, returned as a string.

`Subs` - Bloomberg EMSX order subscription

structure

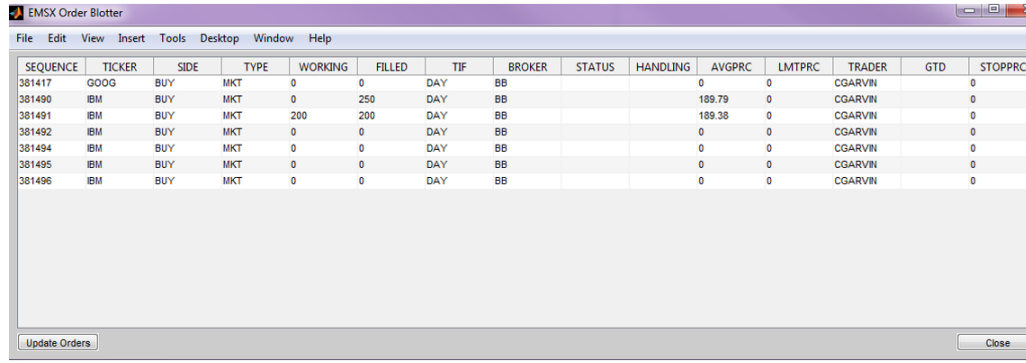
Bloomberg EMSX order subscription, returned as an object.

Examples

Display Order in Order Blotter Interface

Start the EMSX order blotter interface for connection object `C`.

```
emsxOrderBlotter(c)
```



The order blotter interface shows the current order information for a trader.

Define a `reqSruct` and then create a Bloomberg order.

```
reqStruct.EMSX_AMOUNT = int32(330);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
reqStruct.EMSX_TICKER = 'XYZ';
b.createOrderAndRoute(reqStruct,'useDefaultEventHandler',false)
```

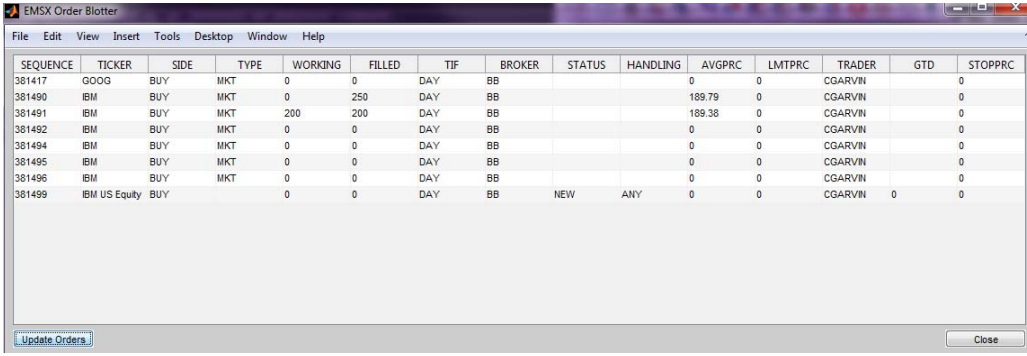
```
 CreateOrderAndRoute = {

     EMSX_SEQUENCE = 381499

     EMSX_ROUTE_ID = 1

     MESSAGE = Order created and routed

     }
```



This updates the order blotter interface with information on the created and routed order (EMSX_SEQUENCE `381499`) using the event handler function `processEventToBlotter`. As orders are created and managed, the blotter is updated.

See Also

`emsx | createOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

processEvent

Purpose

Sample Bloomberg EMSX event handler

Syntax

```
processEvent(c)
```

Description

`processEvent(c)` processes the EMSX event queue associated with Bloomberg EMSX connection handle, `c`.

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

Examples

Continually Process the Bloomberg EMSX Event Queue

Use the following command to continually process the EMSX event queue.

```
T = timer('TimerFcn',{@c.processEvent},'Period',1,'ExecutionMode','fixedRate')
```

See Also

`createOrderAndRoute` | `orders` | `modifyOrder` | `routes` | `deleteOrder` | `routeOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

Purpose

Route Bloomberg EMSX order

Syntax

```
R = routeOrder(c,reqStruct)
R = routeOrder(c,reqStruct,Name,Value)
```

Description

`R = routeOrder(c,reqStruct)` routes a Bloomberg EMSX order and returns a status message using the default event handler.

`R = routeOrder(c,reqStruct,Name,Value)` uses additional options specified by one or more `Name`, `Value` pair arguments. Route a Bloomberg EMSX order using optional name-value arguments to specify a custom event handler or timeout value for the event handler.

Note Name-value pair arguments can be input as a single input structure containing some or all of the property fields, for example:

```
p.timeOut = 1000;
routeOrder(c,reqStruct,p)
```

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

reqStruct - Order request structure

structure

Order request structure, specified using EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `reqStruct`.

Example:

```
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_TIF = 'DAY';
```

```
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
reqStruct.EMSX_SIDE = 'BUY';
```

Data Types
`struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`routeOrder(c,reqStruct,'useDefaultEventHandler',false)`

'useDefaultEventHandler' - Flag for event handler preference

`true` (default) | logical with value `true` or `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, as specified by the string `true` or `false`. When this flag is set to the default, `true`, the default event handler is used. If a custom event handler is used, this flag must be set to `false`.

Example: `'useDefaultEventHandler',false`

Data Types
`logical`

'timeOut' - Connection timeout value for event handler for Bloomberg EMSX service

500 milliseconds (default) | nonnegative integer

Connection timeout value, specified as a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types
`char`

Output Arguments

R - Return status for requested event

structure

Return status for the order event, returned as a structure.

Examples

Route Order for Bloomberg EMSX Using Default Event Handler

Define the `reqstruct` and note that `EMSX_SEQUENCE` must denote an existing order sequence number.

```
reqStruct.EMSX_SEQUENCE = int32(335877);
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
r = routeOrder(c,reqStruct)
```

```
 r =

     EMSX_SEQUENCE: 335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Routed'
```

Route Order for Bloomberg EMSX Using Custom Event Handler

Define the `reqstruct` and note that `EMSX_SEQUENCE` must denote an existing order sequence number.

```
reqStruct.EMSX_SEQUENCE = int32(335877);
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
```

```
routeOrder(c,reqStruct,'useDefaultEventHandler',false)
processEvent(c)

Route = {

     EMSX_SEQUENCE = 335877

     EMSX_ROUTE_ID = 1

     MESSAGE = Order Routed

     }
```

Route Order for Bloomberg EMSX Using `timeOut` Value

Define the `reqstruct` and note that `EMSX_SEQUENCE` must denote an existing order sequence number.

```
reqStruct.EMSX_SEQUENCE = int32(335877);
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
routeOrder(c,reqStruct,'timeOut',200)

r =

     EMSX_SEQUENCE: 335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Routed'
```

See Also

`createOrder` | `createOrderAndRoute` | `orders` | `modifyOrder` | `routes` | `deleteOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

routeOrderWithStrat

Purpose

Route Bloomberg EMSX order with strategies

Syntax

```
R = routeOrderWithStrat(c,reqStruct,stratStruct)
R = routeOrderWithStrat(c,reqStruct,stratStruct,Name,Value)
```

Description

`R = routeOrderWithStrat(c,reqStruct,stratStruct)` routes a Bloomberg EMSX order with strategies and returns the order sequence number, route ID, and status message using the default event handler.

`R = routeOrderWithStrat(c,reqStruct,stratStruct,Name,Value)` uses additional options specified by one or more `Name`, `Value` pair arguments. Route a Bloomberg EMSX order with strategies using optional name-value arguments to specify a custom event handler or timeout value for the event handler.

Note Name-value pair arguments can be input as a single input structure containing some or all of the property fields, for example:

```
p.timeOut = 1000;
routeOrderWithStrat(c,reqStruct,stratStruct,p)
```

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

reqStruct - Order request structure

structure | integer for `EMSX_SEQUENCE` number

Order request structure, specified using EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `reqStruct`.

Note EMSX_SEQUENCE must denote an existing order sequence number.

Example:
```
reqStruct.EMSX_SEQUENCE = int32(335877);
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
```

Data Types
int32 | struct

stratStruct - Order strategies structure
structure

Order strategies structure specified by the elements of the fields EMSX_STRATEGY_FIELD_INDICATORS and EMSX_STRATEGY_FIELDS in the stratStruct. In addition, the field elements of stratStruct must align with the fields for the strategy specified by STRATSTRUCT.EMSX_STRATEGY_NAME. For more information on strategy fields and ordering, see getBrokerInfo.

When using stratStruct, set STRATSTRUCT.EMSX_STRATEGY_FIELD_INDICATORS equal to 0 for each field so that the field data setting in STRATSTRUCT.EMSX_FIELD_DATA is used. Also set STRATSTRUCT.EMSX_STRATEGY_FIELD_INDICATORS equal to 1 to ignore the data in STRATSTRUCT.EMSX_FIELD_DATA.

Example:
```
stratStruct.EMSX_STRATEGY_NAME = 'SSP';
stratStruct.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0
0]);
stratStruct.EMSX_STRATEGY_FIELDS =
{'09:30:00','14:30:00',50};
```

Data Types
struct

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`routeOrderWithStrat(c,reqStruct,stratStruct,'useDefaultEventHandler',fals`

'useDefaultEventHandler' - Flag for event handler preference

`true` (default) | logical with value `true` or `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, as specified by the string `true` or `false`. When this flag is set to the default, `true`, the default event handler is used. If a custom event handler is used, this flag must be set to `false`.

Example: `'useDefaultEventHandler',false`

Data Types
`logical`

'timeOut' - Connection timeout value for event handler for Bloomberg EMSX service

500 milliseconds (default) | nonnegative integer

Connection timeout value, specified as a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types
`char`

Output Arguments

R - Return status for order event

structure

Return status for the order event, returned as a structure.

Examples

Route Bloomberg EMSX Order with Strategies Using Default Event Handler

Define the order request structure and strategies structure and then route the order.

```
reqStruct.EMSX_SEQUENCE = int32(335877);
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
stratStruct.EMSX_STRATEGY_NAME = 'SSP';
stratStruct.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
stratStruct.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
 r = routeOrderWithStrat(c,reqStruct,stratStruct)

 r =

     EMSX_SEQUENCE: 335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Routed'
```

Route Bloomberg EMSX Order with Strategies Using Custom Event Handler

Define the order request structure and strategies structure and then route the order.

```
reqStruct.EMSX_SEQUENCE = int32(335877);
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
stratStruct.EMSX_STRATEGY_NAME = 'SSP';
```

```
stratStruct.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
stratStruct.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
routeOrderWithStrat(c,reqStruct,stratStruct,'useDefaultEventHandler',false)
processEvent(c)
```

```
 Route = {

     EMSX_SEQUENCE = 335877

     EMSX_ROUTE_ID = 1

     MESSAGE = Order Routed

     }
```

Route Bloomberg EMSX Order with Strategies Using `timeOut` Value

Define the order request structure and strategies structure and then route the order.

```
reqStruct.EMSX_SEQUENCE = int32(335877);
reqStruct.EMSX_TICKER = 'XYZ';
reqStruct.EMSX_AMOUNT = int32(100);
reqStruct.EMSX_ORDER_TYPE = 'MKT';
reqStruct.EMSX_BROKER = 'BB';
reqStruct.EMSX_TIF = 'DAY';
reqStruct.EMSX_HAND_INSTRUCTION = 'ANY';
stratStruct.EMSX_STRATEGY_NAME = 'SSP';
stratStruct.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
stratStruct.EMSX_STRATEGY_FIELDS = {'09:30:00','14:30:00',50};
routeOrderWithStrat(c,reqStruct,stratStruct,'timeOut',200)
```

```
 r =

     EMSX_SEQUENCE: 335877
     EMSX_ROUTE_ID: 1
           MESSAGE: 'Order Routed'
```

See Also

`createOrderAndRouteWithStrat` | `getRouteInfo` | `createOrder` | `orders` | `deleteOrder` | `routes` | `routeOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

Purpose

Obtain Bloomberg EMSX route subscription

Syntax

```
R = routes(c,fields)
R = routes(c,fields,Name,Value)
```

Description

`R = routes(c,fields)` subscribes to Bloomberg EMSX fields and returns information about existing routes using the default event handler.

`R = routes(c,fields,Name,Value)` uses additional options specified by one or more `Name`, `Value` pair arguments. Subscribe to Bloomberg EMSX fields and return information about existing routes using optional name-value arguments to specify a custom event handler or timeout value for the event handler.

Note Optional name-value pair arguments can be input as a single input structure containing some or all of the property fields, for example:

```
p.timeOut = 1000;
routes(c,{'EMSX_BROKER','EMSX_WORKING'},p)
```

Input Arguments

c - Connection object for Bloomberg EMSX service

object structure

Connection object for Bloomberg EMSX service, specified using `emsx`.

fields - EMSX field information

cell array

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`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `routes(c, {'EMSX_BROKER','EMSX_WORKING'},'useDefaultEventHandler',false)`

'useDefaultEventHandler' - Flag for event handler preference

`true` (default) | logical with value `true` or `false`

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, as specified by the string `true` or `false`. When this flag is set to the default, `true`, the default event handler is used. If a custom event handler is used, this flag must be set to `false`.

Example: `'useDefaultEventHandler',false`

Data Types
`logical`

'timeOut' - Connection timeout value for event handler for Bloomberg EMSX service

500 milliseconds (default) | nonnegative integer

Connection timeout value, specified as a nonnegative integer in units of milliseconds.

Example: `'timeOut',200`

Data Types
`char`

Output Arguments

R - Return status for requested event

structure

Return status for the route subscription for existing routes, returned as a structure.

Examples

Route Subscription for Bloomberg EMSX Using Default Event Handler

Return the status for route subscription for existing routes.

```
routes(c, {'EMSX_BROKER','EMSX_WORKING'})
```

```
routes =

                    MSG_TYPE: {3x1 cell}
                MSG_SUB_TYPE: {3x1 cell}
                EVENT_STATUS: [3x1 int32]
                  API_SEQ_NUM: [3x1 int64]
               EMSX_SEQUENCE: [3x1 int32]
               EMSX_ROUTE_ID: [3x1 int32]
                EMSX_FILL_ID: [3x1 int32]
                   EMSX_SIDE: {3x1 cell}
                 EMSX_AMOUNT: [3x1 int32]
                 EMSX_FILLED: [3x1 int32]
             EMSX_AVG_PRICE: [3x1 double]
                EMSX_BROKER: {3x1 cell}
               EMSX_WORKING: [3x1 int32]
                EMSX_TICKER: {3x1 cell}
              EMSX_EXCHANGE: {3x1 cell}

                        ...
```

Route Subscription for Bloomberg EMSX Using Custom Event Handler

Return the status for route subscription for existing routes using a custom event handler.

```
routes(c, {'EMSX_BROKER','EMSX_WORKING'},'useDefaultEventHandler',false)
processEvent(c)

OrderRouteFields = {

    MSG_TYPE = E

    MSG_SUB_TYPE = R

    EVENT_STATUS = 4

    API_SEQ_NUM = 1

    EMSX_SEQUENCE = 381490

    EMSX_ROUTE_ID = 1

    EMSX_FILL_ID = 0

   ...
```

Route Subscription for Bloomberg EMSX Using `timeOut` Value

Return the status for route subscription for existing routes using a `timeout` value.

```
routes(c, {'EMSX_BROKER','EMSX_WORKING'},'timeOut',200)

routes =

                    MSG_TYPE: {3x1 cell}
                MSG_SUB_TYPE: {3x1 cell}
                EVENT_STATUS: [3x1 int32]
                  API_SEQ_NUM: [3x1 int64]
               EMSX_SEQUENCE: [3x1 int32]
               EMSX_ROUTE_ID: [3x1 int32]
                EMSX_FILL_ID: [3x1 int32]
                   EMSX_SIDE: {3x1 cell}
```

```
    EMSX_AMOUNT: [3x1 int32]
    EMSX_FILLED: [3x1 int32]
EMSX_AVG_PRICE: [3x1 double]
    EMSX_BROKER: {3x1 cell}
   EMSX_WORKING: [3x1 int32]
    EMSX_TICKER: {3x1 cell}
  EMSX_EXCHANGE: {3x1 cell}

                ...
```

See Also

`emsx` | `createOrderAndRoute` | `deleteRoute` | `modifyRoute` | `routeOrder`

Related Examples

- "Bloomberg EMSX Order Management" on page 3-14
- "Bloomberg EMSX Route Management" on page 3-19
- "Bloomberg EMSX Order and Route Management" on page 3-24

Concepts

- "Workflow for Bloomberg EMSX" on page 2-2

Purpose

Create X_TRADER connection

Syntax

```
X = xtrdr
```

Description

`X = xtrdr` starts X_TRADER or connects to an existing X_TRADER session.

Output Arguments

X - Connection object
object structure

Connection object for X_TRADER session.

Limitations

- You should only create one X_TRADER connection per MATLAB session. To create a new X_TRADER connection, start a new MATLAB session.

Examples

Create a Connection to X_TRADER

```
X = xtrdr
```

```
x =

  xtrdr with properties:

          Gate: [1x1 COM.Xtapi_TTGate_1]
    InstrNotify: []
     Instrument: []
       OrderSet: []
```

See Also

`close`

Related Examples

- “X_TRADER Price Update” on page 3-3
- “X_TRADER Price Update Depth” on page 3-5
- “X_TRADER Order Submission” on page 3-9

Concepts

- “Workflows for Trading Technologies X_TRADER” on page 2-4

External Web Sites

- X_TRADER API

Purpose

Close X_TRADER connection

Syntax

```
close(X)
```

Description

`close(X)` closes the X_TRADER connection `X`.

Input Arguments

X - Connection object

object structure

Connection object for an X_TRADER session.

Examples

Close X_TRADER Connection

```
close(X)
```

See Also

`xtrdr`

Related Examples

- "X_TRADER Price Update" on page 3-3
- "X_TRADER Price Update Depth" on page 3-5
- "X_TRADER Order Submission" on page 3-9

Concepts

- "Workflows for Trading Technologies X_TRADER" on page 2-4

External Web Sites

- X_TRADER API

createInstrument

Purpose

Create instrument for X_TRADER

Syntax

```
createInstrument(X,S)
createInstrument(X,Name,Value)
```

Description

`createInstrument(X,S)` creates the `xtrdr` instrument defined by the structure `S` with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

`createInstrument(X,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 Programming Tutorial* or *X_TRADER API Class Reference*.

Input Arguments

X - Connection object

object structure

Connection object, specified using `xtrdr`.

S - xtrdr input structure

structure

`xtrdr` input structure, specified using fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Note 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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
createInstrument(X,'Exchange','Eurex','Product','OGBM','ProdType','Opt
P12300','Alias','TestInstrument3')
```

'Property1' - Valid X_TRADER API options

string

Valid X_TRADER API options, specified using the details described in Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Requirement: Restart the MATLAB session before reusing an `'Alias'` setting.

Note When using the `'Alias'` property, ensure that every `'Alias'` name is unique across all X_TRADER instruments.

Data Types
`char`

'Property2' - Valid X_TRADER API options

string

Valid X_TRADER API options, specified using the details described in Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Data Types
`char`

Examples

Create an X_TRADER Instrument 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.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.

Note Any symbols in these examples are from an internal MathWorks test license. These symbols do not represent real exchanges. For more information on correct symbols for the instrument, refer to **Market Explorer** in **X_TRADER Pro**.

Create an `xtrdr` instrument.

```
createInstrument(X,S);
```

Close the connection.

```
close(X)
```

Create an X_TRADER Instrument Using Name-Value Pairs

Start X_TRADER.

```
X = xtrdr;
```

Create an `xtrdr` instrument using name-value pairs corresponding to valid X_TRADER API options.

```
createInstrument(X,'Exchange','Eurex','Product','OGBM',...
                'ProdType','Option','Contract','Jan12 P12300',...
                'Alias','TestInstrument3');
```

Requirement: Restart the MATLAB session before reusing an `'Alias'` setting.

Note Any symbols in these examples are from an internal MathWorks test license. These symbols do not represent real exchanges. For more information on correct symbols for the instrument, refer to **Market Explorer** in **X_TRADER Pro**.

Close the connection.

```
close(X)
```

See Also

`xtrdr` | `createNotifier` | `createOrderProfile` | `createOrderSet`

Related Examples

- "X_TRADER Price Update" on page 3-3
- "X_TRADER Price Update Depth" on page 3-5
- "X_TRADER Order Submission" on page 3-9

Concepts

- "Workflows for Trading Technologies X_TRADER" on page 2-4

External Web Sites

- X_TRADER API

Purpose

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 Programming 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 Programming Tutorial* or *X_TRADER API Class Reference*.

Input Arguments

`X` - Connection object

object structure

Connection object, specified 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 Programming 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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
createNotifier(X,'Instrument',[],'UpdateFilter','','EnablePriceUpdates',-
```

'Property1' - Valid X_TRADER API options

string

Valid X_TRADER API options, specified using the details described in Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Example:
```
createNotifier(X,'Instrument',[],'UpdateFilter','','EnablePriceUpdates',-
```

Data Types
`char`

'Property2' - Valid X_TRADER API options

string

Valid X_TRADER API options, specified using the details described in Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Example:
```
createNotifier(X,'Instrument',[],'UpdateFilter','','EnablePriceUpdates',-
```

Data Types
`char`

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.Instrument = [];
S.UpdateFilter = '';
S.EnablePriceUpdates = -1;
S.EnableDepthUpdates = 0;
S.DebugLogLevel = 3;
S.EnableOrderSetUpdates = -1;
S.PriceList = [];
S.DeliverAllPriceUpdates = 0;
S
```

```
S =

               Instrument: []
             UpdateFilter: ''
       EnablePriceUpdates: -1
       EnableDepthUpdates: 0
            DebugLogLevel: 3
    EnableOrderSetUpdates: -1
                PriceList: []
   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,'Instrument',[],'UpdateFilter','',...
               'EnablePriceUpdates',-1,'EnableDepthUpdates',0,...
               'DebugLogLevel',3,'EnableOrderSetUpdates',-1,...
               'PriceList',[],'DeliverAllPriceUpdates',0);
```

Close the connection.

```
close(X)
```

See Also

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

Related Examples

- "X_TRADER Price Update" on page 3-3
- "X_TRADER Price Update Depth" on page 3-5
- "X_TRADER Order Submission" on page 3-9

Concepts

- "Workflows for Trading Technologies X_TRADER" on page 2-4

External Web Sites

- X_TRADER API

Purpose

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 Programming 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 Programming Tutorial* or *X_TRADER API Class Reference*.

Input Arguments

`X` - `xtrdr` connection object

object structure

`xtrdr` connection object, specified 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 Programming 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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
createOrderProfile(X,'Instrument',[],'Customer','<Default>','Alias','','R
```

'Property1' - Valid X_TRADER API options
string

Valid X_TRADER API options, specified using the details described in Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Example:
```
createOrderProfile(X,'Instrument',[],'Customer','<Default>','Alias','','R
```

Data Types
`char`

'Property2' - Valid X_TRADER API options
string

Valid X_TRADER API options, specified using the details described in Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Example:
```
createOrderProfile(X,'Instrument',[],'Customer','<Default>','Alias','','R
```

Data Types
`char`

Output Arguments

P - Order profile

structure

Order profile, returned as a structure.

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)
```

See Also

`xtrdr` | `createInstrument` | `createNotifier` | `createOrderSet`

Related Examples

- "X_TRADER Price Update" on page 3-3
- "X_TRADER Price Update Depth" on page 3-5
- "X_TRADER Order Submission" on page 3-9

Concepts

- "Workflows for Trading Technologies X_TRADER" on page 2-4

External Web Sites

- X_TRADER API

Purpose

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 Programming 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 Programming 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 Programming Tutorial* or *X_TRADER API Class Reference*.

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 Programming 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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
createOrderSet(X,'Count',0,'Alias','','ReadProperties','b','WriteProperti
```

'Property1' - X_TRADER API options
string

X_TRADER API options, specified using the details described in Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Data Types
`char`

'Property2' - X_TRADER API options
string

X_TRADER API options, specified using the details described in Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Data Types
`char`

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)
```

See Also

`xtrdr` | `createInstrument` | `createNotifier` | `createOrderProfile`

Related Examples

- "X_TRADER Price Update" on page 3-3
- "X_TRADER Price Update Depth" on page 3-5
- "X_TRADER Order Submission" on page 3-9

Concepts

- "Workflows for Trading Technologies X_TRADER" on page 2-4

External Web Sites

- X_TRADER API

getData

Purpose

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 Programming 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`.

Input Arguments

`X` - Connection object

object structure

`xtrdr` connection object, specified using `xtrdr`.

`S` - Instrument object

instrument

Instrument object created by `createInstrument` or aliases with fields corresponding to valid X_TRADER API options. For details, see the Trading Technologies *X_TRADER API Programming Tutorial* or *X_TRADER API Class Reference*.

Example: `x.Instrument(1)`

`F` - Fields for the instrument object

string | cell array of strings

Fields for the instrument object or aliases, `S`. `F` without a corresponding `S` are fields for all instruments associated with the `xtrdr` session object, `X`.

Example: `{'Exchange','Last'}`

Data Types
`char | cell`

Output Arguments

D - X_TRADER data

strings

X_TRADER data, returned as strings in MATLAB and missing data is returned as `NaN`.

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}
```

See Also

`xtrdr` | `createInstrument`

Related Examples

- "X_TRADER Price Update" on page 3-3
- "X_TRADER Price Update Depth" on page 3-5
- "X_TRADER Order Submission" on page 3-9

Concepts

- "Workflows for Trading Technologies X_TRADER" on page 2-4

External Web Sites

- X_TRADER API

Purpose

Create CQG connection object

Syntax

```
c = cqg
```

Description

`c = cqg` creates a CQG connection object `c`.

Output Arguments

c - CQG connection

connection object

CQG connection, returned as a CQG connection object. The properties of this object are as follows:

Property	Description
`Handle`	CQG ActiveX object
`APIConfig`	API configuration type library specification

These properties are determined by the CQG API.

Examples

Create the CQG Connection Object

Create the CQG connection object using `cqg`.

```
c = cqg
```

```
c =

  cqg with properties:

      Handle: [1x1 COM.CQG_CQGCEL_4]
   APIConfig: [1x1 Interface.CQG_4.0_Type_Library_-_Revised_API.ICQGA
```

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
```

Close the CQG connection.

```
close(c);
```

See Also

`startUp` | `close`

Related Examples

- "Create CQG Order" on page 3-40
- "Request CQG Historical Data" on page 3-46
- "Request CQG Intraday Tick Data" on page 3-49
- "Request CQG Real-Time Data" on page 3-54

Concepts

- "Workflow for CQG" on page 2-8

External Web Sites

- *CQG API Reference Guide*

Purpose

Close CQG connection

Syntax

```
close(c)
```

Description

`close(c)` closes CQG connection `c`.

Input Arguments

`c` - CQG connection

connection object

CQG connection, specified as a CQG connection object created using `cqg`.

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);
```

See Also

`cqg` | `shutDown`

Related Examples

- "Create CQG Order" on page 3-40
- "Request CQG Historical Data" on page 3-46
- "Request CQG Intraday Tick Data" on page 3-49
- "Request CQG Real-Time Data" on page 3-54

Concepts

- "Workflow for CQG" on page 2-8

External Web Sites

- *CQG API Reference Guide*

createOrder

Purpose

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`.

Input Arguments

c - CQG connection

connection object

CQG connection, specified as a CQG connection object created using `cqg`.

s - CQG instrument name

string | `CQGInstrument` object

CQG instrument name, specified as a string or a `CQGInstrument` object, denoting the instrument or security for the order transaction. For more information about creating a `CQGInstrument` object, see *CQG API Reference Guide*.

Data Types
`char`

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
scalar

CQG order quantity, specified as a 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*.

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 Order" on page 3-40. 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 String

To create and place a market order for shares of an instrument with the CQG Trader Com API using a string 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. Then, set up the account credentials `accountHandle`. For an example demonstrating these activities, see "Create CQG Order" on page 3-40. See *CQG API Reference Guide* to learn more about the event handlers and the API configuration properties.

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 Order" on page 3-40. 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 Order" on page 3-40. 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 Order" on page 3-40. 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);
```

See Also

`cqg` | `history` | `realtime` | `timeseries`

Related Examples

- "Create CQG Order" on page 3-40

Concepts

- "Workflow for CQG" on page 2-8

External Web Sites

- *CQG API Reference Guide*

history

Purpose

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`.

Input Arguments

`c` - CQG connection

connection object

CQG connection, specified as a CQG connection object created using `cqg`.

`s` - CQG instrument name

string

CQG instrument name, specified as a string identifying the instrument or security.

Data Types

`char`

`startdate` - Start date

date string | date scalar

Start date, specified as a starting date string or scalar.

Data Types

`double | char`

`enddate` - End date

date string | date scalar

End date, specified as an ending date string or scalar.

Data Types
`double | char`

period - Bar size

`'hpDaily'` (default) | `'hpWeekly'` | `'hpMonthly'` | `'hpQuarterly'` | `'hpSemiannual'` | `'hpYearly'`

Bar size, specified as one of the above enumerated strings 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`

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 3-46. 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.

```
instrument = 'XYZ.XYZ';
startdate = floor(now) - 10;
enddate = floor(now);
```

```
period = 'hpDaily';

history(c,instrument,startdate,enddate,period);
```

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 3-46. 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`.

```
instrument = 'XYZ.XYZ';
startdate = floor(now) - 10;
enddate = floor(now);
period = 'hpDaily';

history(c,instrument,startdate,enddate,period,x);
```

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` | `createOrder` | `timeseries` | `realtime`

Related Examples

- "Request CQG Historical Data" on page 3-46

Concepts

- "Workflow for CQG" on page 2-8

External Web Sites

- *CQG API Reference Guide*

Purpose

Subscribe to CQG instrument

Syntax

```
realtime(c,s)
```

Description

`realtime(c,s)` subscribes to a CQG instrument `s` using CQG connection `c`.

Input Arguments

c - CQG connection

connection object

CQG connection, specified as a CQG connection object created using `cqg`.

s - CQG instrument name

string

CQG instrument name, specified as a string identifying the instrument or security.

Data Types

`char`

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 3-54. 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);
```

See Also

`cqg` | `createOrder` | `history` | `timeseries`

Related Examples

- “Request CQG Real-Time Data” on page 3-54

Concepts

- “Workflow for CQG” on page 2-8

External Web Sites

- *CQG API Reference Guide*

shutDown

Purpose

Close CQG connection

Syntax

```
shutDown(c)
```

Description

`shutDown(c)` closes the CQG connection `c`.

Input Arguments

`c` - CQG connection

connection object

CQG connection, specified as a CQG connection object created using `cqg`.

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);
```

See Also

`startUp` | `cqg` | `close`

Related Examples

- "Create CQG Order" on page 3-40
- "Request CQG Historical Data" on page 3-46
- "Request CQG Intraday Tick Data" on page 3-49
- "Request CQG Real-Time Data" on page 3-54

Concepts

- "Workflow for CQG" on page 2-8

External Web Sites

- *CQG API Reference Guide*

startUp

Purpose

Create CQG connection

Syntax

```
startUp(c)
```

Description

`startUp(c)` creates the CQG connection `c`.

Input Arguments

c - CQG connection

connection object

CQG connection, specified as a CQG connection object created using `cqg`.

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);
```

See Also

`shutDown` | `cqg` | `close`

Related Examples

- "Create CQG Order" on page 3-40
- "Request CQG Historical Data" on page 3-46
- "Request CQG Intraday Tick Data" on page 3-49
- "Request CQG Real-Time Data" on page 3-54

Concepts

- "Workflow for CQG" on page 2-8

External Web Sites

- *CQG API Reference Guide*

Purpose

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`.

Input Arguments

`c` - CQG connection

connection object

CQG connection, specified as a CQG connection object created using `cqg`.

`s` - CQG instrument name

string

CQG instrument name, specified as a string identifying the instrument or security.

Data Types

`char`

startdate - Start date

date string | date scalar

Start date, specified as a starting date string or scalar.

Data Types
`double | char`

enddate - End date

date string | date scalar

End date, specified as an ending date string or scalar.

Data Types
`double | char`

intraday - Aggregated bar value

scalar | `[]`

Aggregated bar value, specified as a 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`

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 3-49. 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.

```
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 3-49. 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`.

```
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 3-49. 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.

```
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
    0.0007   -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
    0.0007   -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
    ...
```

`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 3-49. 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`.

```
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
    0.0007   -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
    0.0007   -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
    ...
```

`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` | `createOrder` | `history` | `realtime`

Related Examples

- "Request CQG Intraday Tick Data" on page 3-49

Concepts

- "Workflow for CQG" on page 2-8

External Web Sites

- *CQG API Reference Guide*

Purpose

Create IB Trader Workstation connection

Syntax

```
ib = ibtws(host,port)
```

Description

`ib = ibtws(host,port)` creates a connection to IB Trader Workstation on a machine with IP address `host` and port number `port`. `ibtws` returns the IB Trader Workstation connection object `ib`.

Input Arguments

host - IP address of machine where IB Trader Workstation is running

`''` | string for IP address

IP address of the machine where the IB Trader Workstation is running, specified as either an empty string to specify the local machine or an IP address string to specify another machine.

Data Types
`char`

port - IB Trader Workstation port number

scalar

IB Trader Workstation port number, specified as a number designating the connection port of the machine.

Data Types
`double`

Output Arguments

ib - IB Trader Workstation connection

connection object

IB Trader Workstation connection, returned as an IB Trader Workstation connection object. The properties of this object are as follows.

Property	Description
`ClientId`	Application identifier where the connection originated
`Handle`	Interactive Brokers ActiveX object
`Host`	`host` argument
`Port`	`port` argument

These properties are determined by the Interactive Brokers API.

Examples

Connect to the IB Trader Workstation on the Local Machine

Connect to the IB Trader Workstation on the local machine using port number `7496`.

```
ib = ibtws('',7496)
```

```
ib =

  ibtws with properties:

    ClientId: 0
      Handle: [1x1 COM.TWS_TwsCtrl_1]
        Host: ''
        Port: 7496
```

MATLAB returns `ib` as the connection to the IB Trader Workstation with the Interactive Brokers ActiveX object, the local host, and the chosen port number.

Display the `Handle` property of `ib`.

```
ib.Handle
```

```
ans =

    COM.TWS_TwsCtrl_1
```

Close the IB Trader Workstation connection.

```
close(ib);
```

Connect to the IB Trader Workstation on Another Machine

Note The IP address used here is for example purposes only and does not represent a real Interactive Brokers machine.

Use IP address `1111.222.333.44` and port number `7496` to connect to the IB Trader Workstation on another machine.

```
 ib = ibtws('1111.222.333.44',7496)

ib =

  ibtws with properties:

    ClientId: 0
      Handle: [1x1 COM.TWS_TwsCtrl_1]
        Host: '1111.222.333.44'
        Port: 7496
```

MATLAB returns `ib` as the connection to the IB Trader Workstation with the Interactive Brokers ActiveX object, the chosen IP address and the chosen port number.

Display the `Handle` property of `ib`.

```
ib.Handle

ans =

    COM.TWS_TwsCtrl_1
```

Close the IB Trader Workstation connection.

```
close(ib);
```

See Also

`close`

Related Examples

- "Create Interactive Brokers Order" on page 3-30
- "Request Interactive Brokers Historical Data" on page 3-34
- "Stream Interactive Brokers Data" on page 3-36

Concepts

- "Workflow for Interactive Brokers" on page 2-6

External Web Sites

- *Interactive Brokers API Reference Guide*

Purpose

Close IB Trader Workstation connection

Syntax

```
close(ib)
```

Description

`close(ib)` closes the IB Trader Workstation connection `ib`.

Input Arguments

ib - IB Trader Workstation connection

connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

Examples

Close the IB Trader Workstation Connection

Connect to the IB Trader Workstation on the local machine with port number `7496`.

```
ib = ibtws('',7496);
```

`ibtws` creates the IB Trader Workstation connection object `ib`.

Close the IB Trader Workstation connection using the IB Trader Workstation connection object `ib`.

```
close(ib);
```

See Also

`ibtws`

Related Examples

- "Create Interactive Brokers Order" on page 3-30
- "Request Interactive Brokers Historical Data" on page 3-34
- "Stream Interactive Brokers Data" on page 3-36

Concepts

- "Workflow for Interactive Brokers" on page 2-6

External Web Sites

- *Interactive Brokers API Reference Guide*

createOrder

Purpose

Create IB Trader Workstation order

Syntax

```
d = createOrder(ib,ibContract,ibOrder,ibOrderid)
```

Description

`d = createOrder(ib,ibContract,ibOrder,ibOrderid)` creates an IB Trader Workstation order over the IB Trader Workstation connection `ib` using the IB Trader Workstation order object `ibOrder` with a unique order identifier `ibOrderid` to denote the order information. `createOrder` uses the IB Trader Workstation contract object `ibContract` to signify the instrument for the transaction. `createOrder` returns the Interactive Brokers order data object `d` containing data about the completed order.

Input Arguments

ib - IB Trader Workstation connection

connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

ibContract - IB Trader Workstation contract

`IContract` object

IB Trader Workstation contract, specified as an IB Trader Workstation `IContract` object. This object is the instrument or security used in the order transaction. This object is created by calling the Interactive Brokers API function `createContract`. To learn more about `createContract` and the attributes you can set, see *Interactive Brokers API Reference Guide*.

ibOrder - IB Trader Workstation order

`IOrder` object

IB Trader Workstation order, specified as an IB Trader Workstation `IOrder` object. This object contains the order conditions, which are: the action of the order, for example, buy or sell; the order quantity; and the type of order, for example, market or limit. This object is created by calling the Interactive Brokers API function `createOrder`. To learn

more about the attributes you can set and `createOrder`, see *Interactive Brokers API Reference Guide*.

ibOrderid - IB Trader Workstation order unique identifier

scalar

IB Trader Workstation order unique identifier, specified as a scalar.

Data Types
`double`

Output Arguments

d - Interactive Brokers order data

data object

Interactive Brokers order data, returned as a data object containing parameters: status, filled, remaining, average fill price, permanent identifier, parent identifier, last fill price, client identifier, and why held.

Examples

Create the IB Trader Workstation Order

To create an order, set up a connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract`. An `IContract` object is an Interactive Brokers object for containing the necessary data about a security to process transactions. Then, create an IB Trader Workstation `IOrder` object `ibOrder`. An `IOrder` object is an Interactive Brokers object that contains the necessary order conditions to place an order. For an example showing how to create these objects, see "Create Interactive Brokers Order" on page 3-30. See *Interactive Brokers API Reference Guide* to learn more about creating these objects.

Execute the buy market order for two shares with unique identifier `ibOrderid`.

```
ibOrderid = 1;

d = createOrder(ib,ibContract,ibOrder,ibOrderid)

d =
    STATUS: 'Filled'
```

```
    FILLED: 2
    REMAINING: 0
    AVG_FILL_PRICE: 787.5600
    PERM_ID: '1979798454'
    PARENT_ID: 0
    LAST_FILL_PRICE: 787.5600
    CLIENT_ID: 0
    WHY_HELD: ''
```

`d` contains parameters: status, filled, remaining, average fill price, permanent identifier, parent identifier, last fill price, client identifier and why held.

Display the data in the `STATUS` property of `d`.

```
d(1,1).STATUS
```

```
ans =
    Filled
```

Close the IB Trader Workstation connection.

```
close(ib);
```

See Also

`ibtws` | `close` | `history` | `getdata` | `timeseries`

Related Examples

- "Create Interactive Brokers Order" on page 3-30

Concepts

- "Workflow for Interactive Brokers" on page 2-6

External Web Sites

- *Interactive Brokers API Reference Guide*

Purpose

Request current Interactive Brokers data

Syntax

```
d = getdata(ib,ibContract)
```

Description

`d = getdata(ib,ibContract)` requests Interactive Brokers current data asynchronously over the IB Trader Workstation connection `ib` using the IB Trader Workstation contract object `ibContract` to signify the instrument. `getdata` returns Interactive Brokers current data object `d` containing current Interactive Brokers data.

Input Arguments

ib - IB Trader Workstation connection

connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

ibContract - IB Trader Workstation contract

`IContract` object

IB Trader Workstation contract, specified as an IB Trader Workstation `IContract` object. This object is the instrument or security used in the order transaction. This object is created by calling the Interactive Brokers API function `createContract`. To learn more about `createContract` and the attributes you can set, see *Interactive Brokers API Reference Guide*.

Output Arguments

d - Interactive Brokers current data

data object

Interactive Brokers current data, returned as a data object containing tick types last price, last size, volume, bid price, bid size, ask price, and ask size.

Examples

Request Interactive Brokers Current Data

To request current data, set up a connection `ib` using `ibtws` and create an IB Trader Workstation `IContract` object `ibContract` as shown in "Stream Interactive Brokers Data" on page 3-36. An `IContract` object

is an Interactive Brokers object for containing the necessary data about a security to process transactions. See *Interactive Brokers API Reference Guide* to learn more about creating this object.

Request current data using `ibContract`.

```
d = getdata(ib, ibContract)
```

```
d =

    LAST_PRICE: 6.85
     LAST_SIZE: 1.00
        VOLUME: 187.00
     BID_PRICE: 6.84
      BID_SIZE: 14.00
     ASK_PRICE: 6.86
      ASK_SIZE: 13.00
```

`d` contains the last price, last size, volume, bid price, bid size, ask price, and ask size.

Display the data in the `BID_PRICE` property of `d`.

```
d.BID_PRICE
```

```
ans =
    6.84
```

Close the IB Trader Workstation connection.

```
 close(ib);
```

See Also

`ibtws` | `close` | `createOrder` | `history` | `timeseries`

Related Examples

- "Stream Interactive Brokers Data" on page 3-36

Concepts

- "Workflow for Interactive Brokers" on page 2-6

External Web Sites

- *Interactive Brokers API Reference Guide*

history

Purpose

Request Interactive Brokers historical data

Syntax

```
d = history(ib,ibContract,startdate,enddate)
d =
history(ib,ibContract,startdate,enddate,ticktype,period)
```

Description

`d = history(ib,ibContract,startdate,enddate)` requests Interactive Brokers historical data between `startdate` and `enddate` for default tick type `'TRADES'` and default period of `'1 day'` over the IB Trader Workstation connection `ib` using the IB Trader Workstation contract object `ibContract` to signify the instrument. `history` returns a matrix `d` with Interactive Brokers historical data.

`d = history(ib,ibContract,startdate,enddate,ticktype,period)` requests Interactive Brokers historical data for a specific type of market data tick `ticktype` and bar size `period`.

Input Arguments

ib - IB Trader Workstation connection

connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

ibContract - IB Trader Workstation contract

`IContract` object

IB Trader Workstation contract, specified as an IB Trader Workstation `IContract` object. This object is the instrument or security used in the order transaction. This object is created by calling the Interactive Brokers API function `createContract`. To learn more about `createContract` and the attributes you can set, see *Interactive Brokers API Reference Guide*.

startdate - Start date

date string | date scalar

Start date, specified as a starting date string or scalar.

Data Types
`double | char`

enddate - End date
date string | date scalar

End date, specified as an ending date string or scalar.

Data Types
`double | char`

ticktype - Types of market data ticks
`'TRADES'` (default) | `'MIDPOINT'` | `'BID'` | `'ASK'` | `'BID_ASK'` | `'HISTORICAL_VOLATILITY'` | `'OPTION_IMPLIED_VOLATILITY'`

Types of market data ticks, specified as one of the above enumerated strings predetermined by the Interactive Brokers API that denote tick values to collect.

period - Bar size
`'1 day'` (default) | `'1 week'` | `'1 month'`

Bar size, specified as one of the above enumerated strings predetermined by the Interactive Brokers API that denote the periodicity for collecting data.

Output Arguments

d - Interactive Brokers historical data
matrix

Interactive Brokers historical data, returned as a matrix with nine columns: numeric representation of a date, open price, high price, low price, close price, volume, bar count, weighted average price, and flag indicating if there are gaps in the bar.

Examples

Request Interactive Brokers Historical Data with TRADES Default Tick Type and 1-Day Default Period

To request historical data, set up a connection `ib` using `ibtws` and create an IB Trader Workstation `IContract` object `ibContract` as shown in "Request Interactive Brokers Historical Data" on page 3-34. An `IContract` object is an Interactive Brokers object for containing the necessary data about a security to process transactions. See *Interactive Brokers API Reference Guide* to learn more about creating this object.

Request the last 5 days of historical data using `ibContract`.

```
startdate = floor(now) - 5;
enddate = floor(now);

d = history(ib,ibContract,startdate,enddate)
```

```
d =
   1.0e+05 *
    7.3534    0.0079    0.0080    0.0078    0.0078    0.2386    0.1727
    7.3534    0.0078    0.0080    0.0078    0.0079    0.1669    0.1075
    7.3534    0.0079    0.0079    0.0078    0.0078    0.1982    0.1420
    7.3534    0.0079    0.0080    0.0076    0.0078    0.3188    0.2239
    7.3534    0.0078    0.0080    0.0077    0.0080    0.5568    0.3723
```

`d` returns the historical data for 5 days. When `ticktype` and `period` are not specified as input arguments, `history` returns historical data using the default `ticktype` of `'TRADES'` and the default `period` of `'1 day'`.

Each row of `d` contains historical data for 1 day. The columns in matrix `d` are a numeric representation of a date, open price, high price, low price, close price, volume, bar count, weighted average price, and flag indicating if there are gaps in the bar.

Display the open price in matrix `d`.

```
d(1,2)
```

```
ans =
```

```
    790.0000
```

Close the IB Trader Workstation connection.

```
close(ib);
```

Request Interactive Brokers Historical Data with BID Tick Type and 1-Week Period

To request historical data, set up a connection `ib` using `ibtws` and create an IB Trader Workstation `IContract` object `ibContract` as shown in "Request Interactive Brokers Historical Data" on page 3-34. An `IContract` object is an Interactive Brokers object for containing the necessary data about a security to process transactions. See *Interactive Brokers API Reference Guide* to learn more about creating this object.

Request the last 50 days of historical data using `ibContract` with the `'BID'` tick type and a bar size of `'1 week'`.

```
startdate = floor(now) - 50;
enddate = floor(now);
ticktype = 'BID';
period = '1 week';

d = history(ib,ibContract,startdate,enddate,ticktype,period)
```

```
d =
   1.0e+05 *
    7.3529    0.0080    0.0081    0.0078    0.0081   -0.0000   -0.0000
    7.3530    0.0080    0.0084    0.0080    0.0083   -0.0000   -0.0000
    7.3531    0.0082    0.0084    0.0081    0.0081   -0.0000   -0.0000
    7.3532    0.0080    0.0083    0.0079    0.0081   -0.0000   -0.0000
    7.3532    0.0081    0.0082    0.0079    0.0079   -0.0000   -0.0000
    7.3533    0.0079    0.0081    0.0078    0.0078   -0.0000   -0.0000
    7.3534    0.0078    0.0079    0.0077    0.0079   -0.0000   -0.0000
    7.3534    0.0079    0.0080    0.0076    0.0080   -0.0000   -0.0000
```

`d` returns the historical data for 50 days.

Each row of `d` contains historical data for 1 week. The columns in object `d` are a numeric representation of a date, open price, high price, low price, close price, volume, bar count, weighted average price, and flag indicating if there are gaps in the bar.

Display the high price in matrix `d`.

```
d(1,3)
```

```
ans =
    810
```

Close the IB Trader Workstation connection.

```
close(ib);
```

Request Interactive Brokers Historical Data with TRADES Default Tick Type and 1-Month Period

To request historical data, set up a connection `ib` using `ibtws` and create an IB Trader Workstation `IContract` object `ibContract` as shown in "Request Interactive Brokers Historical Data" on page 3-34. An `IContract` object is an Interactive Brokers object for containing the necessary data about a security to process transactions. See *Interactive Brokers API Reference Guide* to learn more about creating this object.

Request the last 50 days of historical data using `ibContract` with the `'TRADES'` default tick type as denoted by the empty string and a bar size of `'1 month'`.

```
startdate = floor(now) - 50;
enddate = floor(now);
ticktype = '';
period = '1 month';

d = history(ib,ibContract,startdate,enddate,ticktype,period)
```

```
d =
   1.0e+05 *
```

```
    7.3529    0.0079    0.0081    0.0078    0.0080    1.9128    1.3384
    7.3532    0.0080    0.0084    0.0079    0.0079    4.0250    2.6757
    7.3534    0.0079    0.0081    0.0076    0.0080    3.6047    2.4843
```

`d` returns the historical data for 50 days.

Each row of `d` contains historical data for 1 month. The columns in object `d` are a numeric representation of a date, open price, high price, low price, close price, volume, bar count, weighted average price, and flag indicating if there are gaps in the bar.

Display the low price in matrix `d`.

```
d(1,4)
```

```
ans =
    780
```

Close the IB Trader Workstation connection.

```
close(ib);
```

See Also

`ibtws` | `close` | `createOrder` | `getdata` | `timeseries`

Related Examples

- "Request Interactive Brokers Historical Data" on page 3-34

Concepts

- "Workflow for Interactive Brokers" on page 2-6

External Web Sites

- *Interactive Brokers API Reference Guide*

timeseries

Purpose

Request Interactive Brokers aggregated intraday data

Syntax

```
d = timeseries(ib,ibContract,startdate,enddate,barsize)
d =
timeseries(ib,ibContract,startdate,enddate,barsize,ticktype)
```

Description

`d = timeseries(ib,ibContract,startdate,enddate,barsize)` requests Interactive Brokers aggregated intraday data between `startdate` and `enddate` with tick aggregation interval `barsize` for default tick type `'TRADES'` over the IB Trader Workstation connection `ib` using IB Trader Workstation contract object `ibContract` to signify the instrument. `timeseries` returns the matrix `d` with Interactive Brokers aggregated intraday data.

`d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype)` requests Interactive Brokers aggregated intraday data for a specific type of market data tick `ticktype`.

Input Arguments

ib - IB Trader Workstation connection

connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtws`.

ibContract - IB Trader Workstation contract

`IContract` object

IB Trader Workstation contract, specified as an IB Trader Workstation `IContract` object. This object is the instrument or security used in the order transaction. This object is created by calling the Interactive Brokers API function `createContract`. To learn more about `createContract` and the attributes you can set, see *Interactive Brokers API Reference Guide*.

startdate - Start date

date string | date scalar

Start date, specified as a starting date string or scalar.

Data Types
`double | char`

enddate - End date
date string | date scalar

End date, specified as an ending date string or scalar.

Data Types
`double | char`

barsize - Tick aggregation interval
`'10 secs' | '15 secs' | '30 secs' | '1 min' | '2 mins' | '3 mins' | ...`

Tick aggregation interval, specified as one of the following enumerated strings predetermined by the Interactive Brokers API that denote the size of aggregated bars for collecting data.

- `'10 secs'`
- `'15 secs'`
- `'30 secs'`
- `'1 min'`
- `'2 mins'`
- `'3 mins'`
- `'5 mins'`
- `'10 mins'`
- `'15 mins'`
- `'20 mins'`
- `'30 mins'`
- `'1 hour'`

- '2 hours'
- '3 hours'
- '4 hours'
- '8 hours'

ticktype - Types of market data ticks
'TRADES' (default) | 'MIDPOINT' | 'BID' | 'ASK' | 'BID_ASK' | 'HISTORICAL_VOLATILITY' | 'OPTION_IMPLIED_VOLATILITY'

Types of market data ticks, specified as one of the above enumerated strings predetermined by the Interactive Brokers API that denote tick values to collect.

Output Arguments

d - Interactive Brokers aggregated intraday data
matrix

Interactive Brokers aggregated intraday data, returned as a matrix with nine columns: a numeric representation of a date, open price, high price, low price, close price, volume, bar count, weighted average price, and flag indicating if there are gaps in the bar.

Examples

Request Interactive Brokers Intraday Data Aggregated Every 5 Minutes with TRADES Default Tick Type

To request intraday data, set up a connection `ib` using `ibtws` and create an IB Trader Workstation `IContract` object `ibContract` as shown in "Stream Interactive Brokers Data" on page 3-36. An `IContract` object is an Interactive Brokers object for containing the necessary data about a security to process transactions. See the *Interactive Brokers API Reference Guide* to learn more about creating this object.

Request intraday data aggregated every 5 minutes using `ibContract`.

```
startdate = floor(now);
enddate = now;
barsize = '5 mins';
```

```
d = timeseries(ib,ibContract,startdate,enddate,barsize)

d =

     735329.40          6.91          6.91          6.85          6.85
     735329.40          6.85          6.87          6.85          6.87
     735329.40          6.87          6.89          6.87          6.87
...
```

`d` returns the aggregated 5-minute data with default tick type `'TRADES'`.

Each row in matrix `d` represents a 5-minute interval. The columns in matrix `d` are a numeric representation of a date, open price, high price, low price, close price, volume, bar count, weighted average price, and flag indicating if there are gaps in the bar.

Display the open price in matrix `d`.

```
d(1,2)

ans =
    6.91
```

Close the IB Trader Workstation connection.

```
close(ib);
```

Request Interactive Brokers Intraday Data Aggregated Every 10 Minutes with a BID Tick Type

To request intraday data, set up a connection `ib` using `ibtws` and create an IB Trader Workstation `IContract` object `ibContract` as shown in "Stream Interactive Brokers Data" on page 3-36. An `IContract` object is an Interactive Brokers object for containing the necessary data about a security to process transactions. See *Interactive Brokers API Reference Guide* to learn more about creating this object.

Request intraday data aggregated every 10 minutes using `ibContract` and `'BID'` tick type.

```
startdate = floor(now);
enddate = now;
barsize = '10 mins';
ticktype = 'BID';

d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype)

d =

     735329.17          6.38          6.38          6.38          6.38
     735329.17          6.38          6.38          6.38          6.38
     735329.18          6.38          6.38          6.38          6.38
...
```

`d` returns the aggregated 10-minute data for `'BID'` tick type.

Each row in matrix `d` represents a 10-minute interval. The columns in matrix `d` are a numeric representation of a date, open price, high price, low price, close price, volume, bar count, weighted average price, and flag indicating if there are gaps in the bar.

Display the high price in matrix `d`.

```
d(1,3)

ans =
    6.38
```

Close the IB Trader Workstation connection.

```
close(ib);
```

See Also

`ibtws` | `close` | `createOrder` | `getdata` | `history`

Related Examples

- “Stream Interactive Brokers Data” on page 3-36

Concepts

- “Workflow for Interactive Brokers” on page 2-6

External Web Sites

- *Interactive Brokers API Reference Guide*