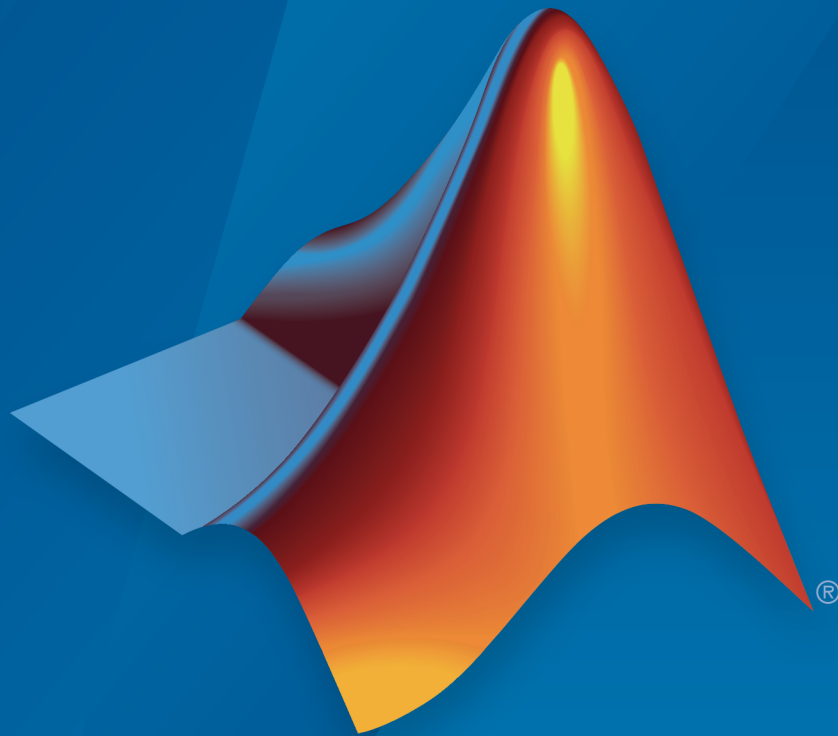


Trading Toolbox™

User's Guide



MATLAB®

R2015a

 MathWorks®

# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*Trading Toolbox™ User's Guide*

© COPYRIGHT 2013–2015 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

## Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://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](http://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)
October 2014	Online only	Revised for Version 2.1.1 (Release 2014b)
March 2015	Online only	Revised for Version 2.2 (Release 2015a)

## Getting Started

<b>Trading Toolbox Product Description</b> .....	1-2
Key Features .....	1-2
<b>Installation</b> .....	1-3
Bloomberg .....	1-3
CQG .....	1-3
Interactive Brokers .....	1-3
Trading Technologies .....	1-3
<b>Trading System Providers</b> .....	1-4
Supported Providers .....	1-4
Connection Requirements .....	1-4
<b>Create an Order Using IB Trader Workstation</b> .....	1-6
<b>Create an Order Using CQG</b> .....	1-9
<b>Create an Order Using Bloomberg EMSX</b> .....	1-11
<b>Create an Order Using X_TRADER</b> .....	1-14
<b>Writing and Running Custom Event Handler Functions with Bloomberg EMSX</b> .....	1-17
Write a Custom Event Handler Function .....	1-17
Run a Custom Event Handler Function .....	1-17
Workflow for Custom Event Handler Function .....	1-18
<b>Writing and Running Custom Event Handler Functions with Interactive Brokers</b> .....	1-20
Write a Custom Event Handler Function .....	1-20
Run a Custom Event Handler Function .....	1-20
Workflow for Custom Event Handler Function .....	1-21

**2**

<b>Workflow for Bloomberg EMSX</b> .....	<b>2-2</b>
<b>Workflows for Trading Technologies X_TRADER</b> .....	<b>2-4</b>
<b>Workflow for Interactive Brokers</b> .....	<b>2-6</b>
Request Interactive Brokers Market Data .....	<b>2-6</b>
Create Interactive Brokers Orders .....	<b>2-7</b>
Request Interactive Brokers Informational Data .....	<b>2-7</b>
<b>Workflow for CQG</b> .....	<b>2-8</b>

**Sample Code for Workflows**

**3**

<b>X_TRADER Workflows</b> .....	<b>3-2</b>
<b>X_TRADER Price Update</b> .....	<b>3-3</b>
<b>X_TRADER Price Update Depth</b> .....	<b>3-5</b>
<b>X_TRADER Order Submission</b> .....	<b>3-9</b>
<b>Create and Manage a Bloomberg EMSX Order</b> .....	<b>3-13</b>
<b>Create and Manage a Bloomberg EMSX Route</b> .....	<b>3-17</b>
<b>Manage a Bloomberg EMSX Order and Route</b> .....	<b>3-22</b>
<b>Create Interactive Brokers Order</b> .....	<b>3-27</b>
<b>Request Interactive Brokers Historical Data</b> .....	<b>3-33</b>
<b>Request Interactive Brokers Real-Time Data</b> .....	<b>3-36</b>
<b>Create Interactive Brokers Combination Order</b> .....	<b>3-40</b>

<b>Create CQG Order</b> .....	<b>3-45</b>
<b>Request CQG Historical Data</b> .....	<b>3-50</b>
<b>Request CQG Intraday Tick Data</b> .....	<b>3-53</b>
<b>Request CQG Real-Time Data</b> .....	<b>3-57</b>

## **Functions — Alphabetical List**

**4**



# Getting Started

---

- “Trading Toolbox Product Description” on page 1-2
- “Installation” on page 1-3
- “Trading System Providers” on page 1-4
- “Create an Order Using IB Trader Workstation” on page 1-6
- “Create an Order Using CQG” on page 1-9
- “Create an Order Using Bloomberg EMSX” on page 1-11
- “Create an Order Using X\_TRADER” on page 1-14
- “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17
- “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20

## 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<sup>®</sup>, 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<sup>®</sup> EMSX, CQG<sup>®</sup> Integrated Client, Interactive Brokers<sup>®</sup> TWS, and Trading Technologies<sup>®</sup> X\_TRADER<sup>®</sup>.

### 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



# Installation

**In this section...**

“Bloomberg” on page 1-3

“CQG” on page 1-3

“Interactive Brokers” on page 1-3

“Trading Technologies” on page 1-3

## Bloomberg

Find the latest installation files at <http://www.bloomberg.com> to install Bloomberg EMSX from Bloomberg L.P. You need a Bloomberg license to install and run Bloomberg EMSX.

## CQG

Find the latest installation files at <http://www.cqg.com> to install CQG. You need a CQG license to install and run CQG.

## Interactive Brokers

- 1 Download and install the IB Trader Workstation<sup>SM</sup> Desktop Trading Client. Find the latest installation files at <https://www.interactivebrokers.com/en/index.php?f=552>.
- 2 Download and install the Interactive Brokers API software. Find the latest installation files at <http://interactivebrokers.github.io/>.

You need an Interactive Brokers license to install and run Interactive Brokers.

## Trading Technologies

Find the latest installation files at <http://www.tradingtechnologies.com> to install Trading Technologies. You need a Trading Technologies license to install and run Trading Technologies.

## Trading System Providers

<b>In this section...</b>
“Supported Providers” on page 1-4
“Connection Requirements” on page 1-4

### 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<sup>®</sup> 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 IB Trader Workstation

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.

```
id = orderid(ib);  
  
result = createOrder(ib,ibContract,ibMktOrder,id)  
  
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

`close` | `createOrder` | `getdata` | `history` | `ibtw`

## External Web Sites

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

## Create an Order Using CQG

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

### Connect to CQG.

```
c = cqg;
```

### Establish event handlers.

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

```
startUp(c)

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

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

### Subscribe to 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

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

## External Web Sites

- <http://cQG.com/Products/CQG-API/CQG-Trader-API.aspx>



## Create an Order Using Bloomberg EMSX

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

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

### Connect to Bloomberg EMSX

- 1 If you are using `emsx` for the first time, you need to install a Java<sup>®</sup> archive file from Bloomberg for `emsx` and other Bloomberg commands to work correctly.

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

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

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

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

- 2 Connect to the Bloomberg EMSX test service.

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

```
c =
```

```
emsx with properties:
```

```
Session: [1x1 com.bloomberglp.blpapi.Session]
Service: [1x1 com.bloomberglp.blpapi.impl.aQ]
```

```
    Ippaddress: 'localhost'  
    Port: 8194
```

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

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

## Create the market order request

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

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

## Create and route the market order

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

```
events = createOrderAndRoute(c,order);  
  
events =  
  
    EMSX_SEQUENCE: 335877  
    EMSX_ROUTE_ID: 1  
    MESSAGE: 'Order created and routed'
```

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

- Bloomberg EMSX order number
- Bloomberg EMSX route identifier

- Bloomberg EMSX message

### **Close the Bloomberg EMSX connection**

`close(c)`

### **See Also**

`close` | `createOrderAndRoute` | `emsx`

## Create an Order Using X\_TRADER

This example shows how to connect to Trading Technologies X\_TRADER and create a market order.

### Connect to Trading Technologies X\_TRADER.

```
c = xtrdr;
```

### Create 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.

```

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

```

**Enable order submission.**

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

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

**Build 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

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

## External Web Sites

- [https://developer.tradingtechnologies.com/x\\_trader-api](https://developer.tradingtechnologies.com/x_trader-api)

# Writing and Running Custom Event Handler Functions with Bloomberg EMSX

## In this section...

“Write a Custom Event Handler Function” on page 1-17

“Run a Custom Event Handler Function” on page 1-17

“Workflow for Custom Event Handler Function” on page 1-18

## Write a Custom Event Handler Function

You can process events related to any Bloomberg EMSX orders and routes by writing a custom event handler function to use with Trading Toolbox. For example, you can plot the changes in the number of shares routed. Follow these tasks to write a custom event handler.

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

For details, see “Create Functions in Files”. For a code example of an event handler function, see the function `processEventToBlotter` in the `emsxOrderBlotter.m` file.

## Run a Custom Event Handler Function

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

- 1 Run `timer` to execute `eventhandler`. The name-value pair argument `TimerFcn` specifies the event handler function. The name-value pair argument `Period` specifies a 1-second delay between executions of the event handler function. When the name-value pair argument `ExecutionMode` is set to `fixedRate`, the event

handler function executes immediately after it is added to the MATLAB execution queue.

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

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

```
start(t)
```

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

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

- 4 Stop the timer to stop data updates.

```
stop(t)
```

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

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

```
delete(t)
```

## Workflow for Custom Event Handler Function

This workflow summarizes the tasks to work with a custom event handler function using Bloomberg EMSX.

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



- 9 Close the connection using `close`.

### **See Also**

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

### **Related Examples**

- “Create Functions in Files”

### **More About**

- “What Is a Function Handle?”

## Writing and Running Custom Event Handler Functions with Interactive Brokers

### In this section...

“Write a Custom Event Handler Function” on page 1-20

“Run a Custom Event Handler Function” on page 1-20

“Workflow for Custom Event Handler Function” on page 1-21

### Write a Custom Event Handler Function

You can process events related to any Interactive Brokers data updates by writing a custom event handler function to use with Trading Toolbox. For example, you can request data about all open orders or retrieve account information. Follow these tasks to write a custom event handler.

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

For details, see “Create Functions in Files”. For a code example of an Interactive Brokers event handler function, see `ibExampleEventHandler.m`.

### Run a Custom Event Handler Function

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

For example, suppose you want to retrieve real-time data from Interactive Brokers using `realtime` with the custom event handler function named `eventhandler`. You can use either of these syntaxes to run `eventhandler`. This code assumes a IB Trader Workstation connection `ib`, IB Trader Workstation `IContract` object `ibContract`, and Interactive Brokers fields `f`.

Use a string.

```
tickerid = realtime(ib,ibContract,f,'eventhandler');
```

Or, use a function handle.

```
tickerid = realtime(ib,ibContract,f,@eventhandler);
```

## Workflow for Custom Event Handler Function

This workflow summarizes the tasks to work with a custom event handler function using Interactive Brokers.

- 1 Write a custom event handler function and save it to a file.
- 2 Create a connection to the IB Trader Workstation using `ibtws`.
- 3 Run an existing function to receive data updates. Use the custom event handler function as an input argument.

---

**Caution:** To run default event handler and sample event handler functions, you must run one event handler function at a time. After you run one event handler, close the IB Trader Workstation connection. Then, create another connection to run a different event handler with the same existing function. Otherwise, MATLAB assigns multiple existing functions to events and errors occur.

---

- 4 Close the connection to the IB Trader Workstation using `close`.

## See Also

`close` | `ibtws` | `realtime`

## More About

- “Create Functions in Files”
- “What Is a Function Handle?”



# Workflow Models

---

- “Workflow for Bloomberg EMSX” on page 2-2
- “Workflows for Trading Technologies X\_TRADER” on page 2-4
- “Workflow for Interactive Brokers” on page 2-6
- “Workflow for CQG” on page 2-8

# Workflow for Bloomberg EMSX

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

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

## **Related Examples**

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

# 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”
- “X\_TRADER Price Update Depth”
- “X\_TRADER Order Submission”

## Workflow for Interactive Brokers

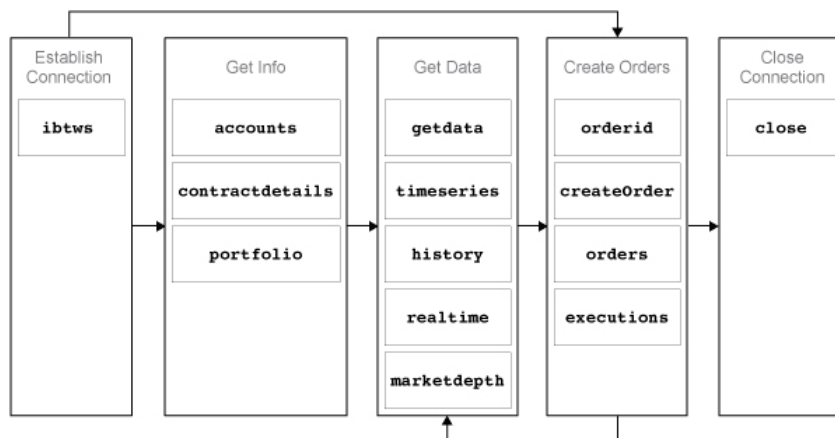
### In this section...

“Request Interactive Brokers Market Data” on page 2-6

“Create Interactive Brokers Orders” on page 2-7

“Request Interactive Brokers Informational Data” on page 2-7

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



### Request Interactive Brokers Market Data

To request current, intraday, real-time, historical, or market depth data:

- 1 Connect to the IB Trader Workstation using `ibtws`.
- 2 Create the IB Trader Workstation `IContract` object.
- 3 Request current data for a security using `getdata`.
- 4 Request intraday data for a security using `timeseries`.
- 5 Request real-time data for a security using `realtime`.
- 6 Request historical data for a security using `history`.
- 7 Request market depth data for a security using `marketdepth`.

- 8 Close the IB Trader Workstation connection using `close`.

## **Create Interactive Brokers Orders**

To submit orders to the IB Trader Workstation:

- 1 Connect to the IB Trader Workstation using `ibtws`.
- 2 Create the IB Trader Workstation `IContract` object.
- 3 Create the IB Trader Workstation `IOrder` object.
- 4 Request a unique order identifier using `orderid`.
- 5 Create and submit the order using `createOrder`.
- 6 Request open order data using `orders`.
- 7 Request executed order data using `executions`.
- 8 Close the IB Trader Workstation connection using `close`.

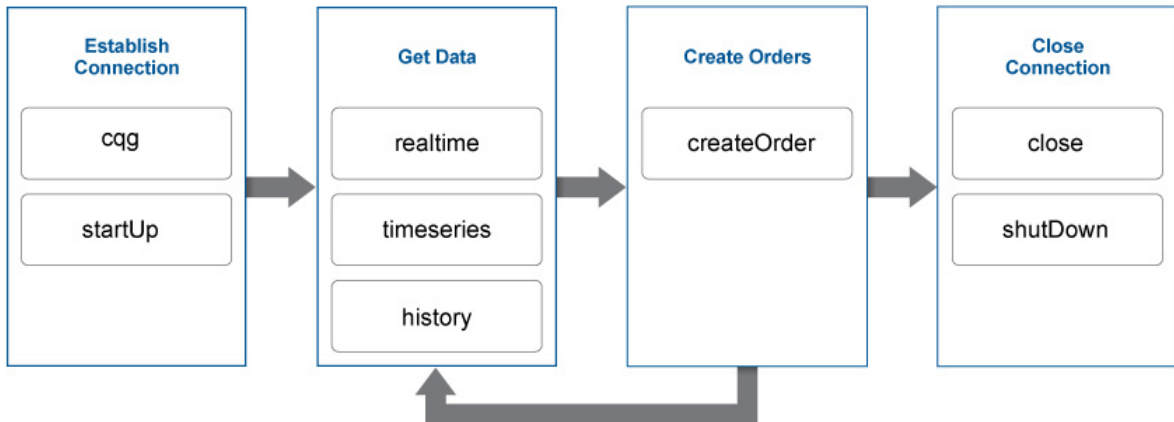
## **Request Interactive Brokers Informational Data**

To request information from the IB Trader Workstation:

- 1 Connect to the IB Trader Workstation using `ibtws`.
- 2 Create the IB Trader Workstation `IContract` object.
- 3 Request contract detailed data using `contractdetails`.
- 4 Request account information using `accounts`.
- 5 Request portfolio data using `portfolio`.
- 6 Close the IB Trader Workstation connection using `close`.

### 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 `cqq`.
- 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 `cqq`.
- 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”
- “Request CQG Historical Data”
- “Request CQG Intraday Tick Data”
- “Request CQG Real-Time Data”



# Sample Code for Workflows

---

- “X\_TRADER Workflows” on page 3-2
- “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
- “Create and Manage a Bloomberg EMSX Order” on page 3-13
- “Create and Manage a Bloomberg EMSX Route” on page 3-17
- “Manage a Bloomberg EMSX Order and Route” on page 3-22
- “Create Interactive Brokers Order” on page 3-27
- “Request Interactive Brokers Historical Data” on page 3-33
- “Request Interactive Brokers Real-Time Data” on page 3-36
- “Create Interactive Brokers Combination Order” on page 3-40
- “Create CQG Order” on page 3-45
- “Request CQG Historical Data” on page 3-50
- “Request CQG Intraday Tick Data” on page 3-53
- “Request CQG Real-Time Data” on page 3-57

## **X\_TRADER Workflows**

X\_TRADER supports the following workflows:

- “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



## 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.

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

### Monitor Events

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

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

The last command tells X\_TRADER to start monitoring the attached instruments using the specified event settings.

### **Close the Connection**

```
close(X)
```

### **See Also**

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

### **Related Examples**

- “X\_TRADER Price Update Depth” on page 3-5
- “X\_TRADER Order Submission” on page 3-9

### **More About**

- “Workflows for Trading Technologies X\_TRADER”

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

### Attach an Instrument to a Notifier

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

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

### Define Events

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

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

### Set Up the Figure Window

Set up the figure window to display depth data.

```
f = figure('Numbertitle', 'off', 'Tag', 'TTPriceUpdateDepthFigure', ...
```

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

#### Create Controls

Create controls for the last price data.

```
bspc = 5;
bwid = 80;
bhgt = 20;

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

#### Create a Table

Create a table containing order information.

```

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

```

### Store Data

```

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

```

### Listen for Event Data

Listen for event data with depth updates enabled.

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

	Bid	Bid Size	Ask	Ask Size
1	46	20	55	15

Exchange	Product	Type	Contract
CME	2F	FUTURE	2F May13

Last Price	Last Qty	Change
51	20	5

The last command instructs X\_TRADER to start monitoring the attached instruments using the specified event settings.

### Close the Connection

```
close(X)
```

### See Also

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

### Related Examples

- “X\_TRADER Price Update” on page 3-3
- “X\_TRADER Order Submission” on page 3-9

### More About

- “Workflows for Trading Technologies X\_TRADER”

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

### Create an Order Set

The `OrderSet` object sends orders to X\_TRADER.

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

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

### Set Position Limit Checks

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

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

### Set a Callback Function

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

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

### Enable Order Submission

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

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

### Build an Order Profile

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

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

### Sample: Create a Market Order

Create a market order to buy 100 shares.

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

### Sample: Create a Limit Order

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

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

### Sample: Create a Stop Market Order

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

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



**Sample: Create a Stop Limit Order**

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

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

**Check the Order Server Status**

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

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

**Verify the Order Server Availability**

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

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

**Close the Connection**

```
close(X)
```

**See Also**

close | createInstrument | createOrderProfile | createOrderSet | xtrdr

**Related Examples**

- “X\_TRADER Price Update” on page 3-3
- “X\_TRADER Price Update Depth” on page 3-5

## **More About**

- “Workflows for Trading Technologies X\_TRADER”

## Create and Manage a Bloomberg EMSX Order

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

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

### Connect to Bloomberg EMSX

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

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

c =

    emsx with properties:

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

    SessionConnectionUp = {
        server = localhost/127.0.0.1:8194
    }

    SessionStarted = {
    }

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

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

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

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

#### Set Up the Order Subscription

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

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

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

```
events =
```

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

```
subs =
```

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

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

#### Create the Order

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

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

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

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

```
order_events =
```

```
EMSX_SEQUENCE: 354646
```

```
MESSAGE: 'Order created'
```

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

- Bloomberg EMSX order number
- Bloomberg EMSX message

### Modify the Order

Define the structure `modorder` that contains these fields:

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

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

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

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

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

```
events =
```

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

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

- Bloomberg EMSX order number
- Bloomberg EMSX message

### Delete the Order

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

```
ordernum.EMSX_SEQUENCE = 354646;
events = deleteOrder(c,ordernum)
events =
    STATUS: '0'
    MESSAGE: 'Order deleted'
```

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

- Bloomberg EMSX status
- Bloomberg EMSX message

### Stop the Order Subscription

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

```
b.Session.unsubscribe(subs)
```

### Close the Bloomberg EMSX Connection

```
close(c)
```

### See Also

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

### Related Examples

- “Create and Manage a Bloomberg EMSX Route” on page 3-17
- “Manage a Bloomberg EMSX Order and Route” on page 3-22

### More About

- “Workflow for Bloomberg EMSX”

## Create and Manage a Bloomberg EMSX Route

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

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

### Connect to Bloomberg EMSX

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

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

c =

    emsx with properties:

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

    SessionConnectionUp = {
        server = localhost/127.0.0.1:8194
    }

    SessionStarted = {
    }

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

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

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

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

#### Set Up the Route Subscription

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

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

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

events =

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

subs =

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

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

#### Create and Route the Order

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

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

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

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



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

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

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

### Modify the Route

Define the `modroute` structure that contains these fields:

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

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

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

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

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

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

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

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

#### Delete the Modified Route

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

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

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

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

```
events =
```

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

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

- Bloomberg EMSX status
- Bloomberg EMSX message

#### Stop the Route Subscription

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

```
b.Session.unsubscribe(subs)
```

#### Close the Bloomberg EMSX Connection

```
close(c)
```

#### See Also

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

## **Related Examples**

- “Create and Manage a Bloomberg EMSX Order” on page 3-13
- “Manage a Bloomberg EMSX Order and Route” on page 3-22

## **More About**

- “Workflow for Bloomberg EMSX”

## Manage a Bloomberg EMSX Order and Route

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

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

### Connect to Bloomberg EMSX

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

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

c =

    emsx with properties:

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

    SessionConnectionUp = {
        server = localhost/127.0.0.1:8194
    }

    SessionStarted = {
    }

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

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

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

- Port number of the machine running the Bloomberg EMSX test service

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

### Set Up the Order and Route Subscription

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

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

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

```
events =
```

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

```

```
osubs =
```

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

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

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

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

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

```
events =
```

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

```

```
rsubs =
```

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

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

#### Create and Route the Order

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

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

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

```
events = createOrderAndRoute(c,order)

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

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

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

#### Modify the Route

Define the `modroute` structure that contains these fields:

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

- Bloomberg EMSX route identifier `EMSX_ROUTE_ID`

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

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

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

```
events = modifyRoute(c,modroute)

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

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

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

### Delete the Route

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

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

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

```
events = deleteRoute(c,routenum)

events =
    STATUS: '1'
```

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

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

- Bloomberg EMSX status
- Bloomberg EMSX message

### Stop the Order and Route Subscription

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

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

### Close the Bloomberg EMSX Connection

```
close(c)
```

### See Also

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

### Related Examples

- “Create and Manage a Bloomberg EMSX Order” on page 3-13
- “Create and Manage a Bloomberg EMSX Route” on page 3-17

### More About

- “Workflow for Bloomberg EMSX”



## Create Interactive Brokers Order

This example shows how to connect to the IB Trader Workstation, request open order data, create an IB Trader Workstation `IContract` object, create an IB Trader Workstation `IOrder` object, and execute the order. For details about the `IContract` and `IOrder` objects, see Interactive Brokers API Reference Guide.

This example uses the sample event handler function `ibExampleOrderEventHandler` to populate an order blotter figure with Interactive Brokers order information. Use this event handler or write a custom event handler function. For details, see “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20.

To access the code for this example, see `IBOrderWorkflow.m`.

### Connect to the IB Trader Workstation

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

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

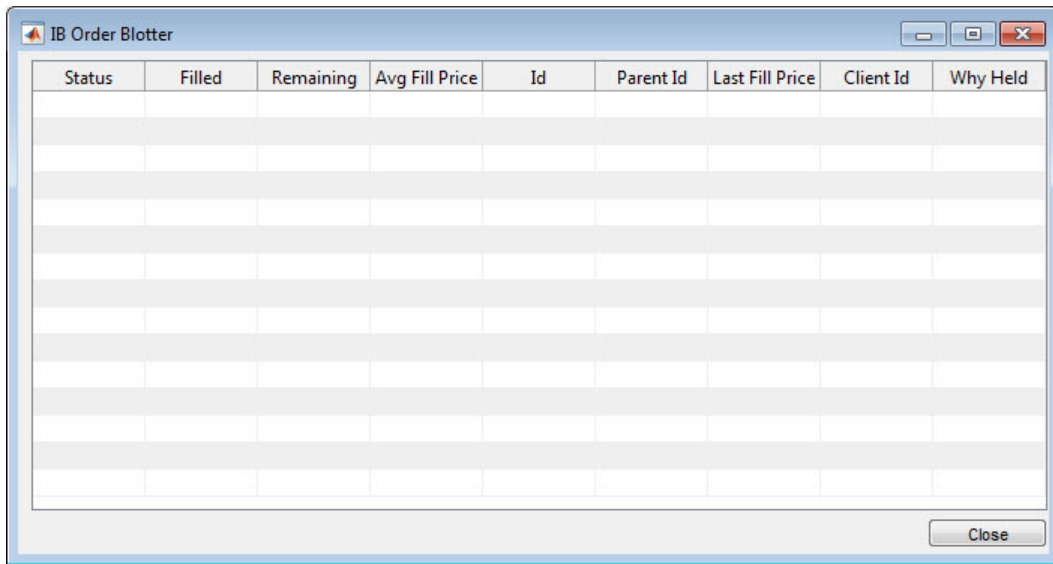
### Create an Example Order Blotter

Create an example order blotter that the event handler populates.

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

```
f = findobj('Tag','IBOrderBlotter');
if isempty(f)
    f = figure('Tag','IBOrderBlotter','MenuBar','none',...
        'NumberTitle','off','Name','IB Order Blotter')
    pos = f.Position;
    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 IB Order Blotter.



The screenshot shows a window titled "IB Order Blotter" with a standard Windows-style title bar (minimize, maximize, close buttons). The main content is a table with the following columns: Status, Filled, Remaining, Avg Fill Price, Id, Parent Id, Last Fill Price, Client Id, and Why Held. The table is currently empty, showing only the header row. A "Close" button is located in the bottom right corner of the window.

Status	Filled	Remaining	Avg Fill Price	Id	Parent Id	Last Fill Price	Client Id	Why Held
--------	--------	-----------	----------------	----	-----------	-----------------	-----------	----------

#### Request Open Order Data

Request information for all open orders using only this client and the sample event handler `ibExampleOrderEventHandler`.

```
o = orders(ib,true,@ibExampleOrderEventHandler);
```

`o` is an empty double because `ibExampleOrderEventHandler` displays the data for all open orders in the IB Order Blotter.



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.

#### **Create the Interactive Brokers Order**

Obtain the next valid order identification number using IB Trader Workstation connection `ib`.

```
id = orderid(ib);
```

Execute the buy market order for two shares using the unique order identifier `id` and sample event handler `ibExampleOrderEventHandler`.

```
createOrder(ib,ibContract,ibOrder,id,@ibExampleOrderEventHandler)
```

MATLAB displays order information in the IB Order Blotter. The IB Order Blotter shows the open order and the filled order.



### Related Examples

- “Create Interactive Brokers Combination Order” on page 3-40
- “Request Interactive Brokers Historical Data”
- “Request Interactive Brokers Real-Time Data”

### More About

- “Workflow for Interactive Brokers”
- “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20

### External Web Sites

- Interactive Brokers API Reference Guide

## Request Interactive Brokers Historical Data

This example shows how to connect to the IB Trader Workstation, create an IB Trader Workstation `IContract` object, and request historical data. For details about the `IContract` object, see Interactive Brokers API Reference Guide. To access the code for this example, see `IBHistoricalDataWorkflow.m`.

### Connect to the IB Trader Workstation and Create the `IContract` Object

Connect to the 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<sup>®</sup> object, the local host, and the port number that you choose.

Create the IB Trader Workstation `IContract` object `ibContract`. Here, this object describes a security with these property values:

- XYZ symbol
- Stock security type
- Aggregate exchange
- USD currency

```
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.

### 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    0.0079    0
    7.3534    0.0078    0.0080    0.0078    0.0079    0.1669    0.1075    0.0079    0
    7.3534    0.0079    0.0079    0.0078    0.0078    0.1982    0.1420    0.0078    0
    7.3534    0.0079    0.0080    0.0076    0.0078    0.3188    0.2239    0.0077    0
    7.3534    0.0078    0.0080    0.0077    0.0080    0.5568    0.3723    0.0079    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:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

#### Close the Connection

Close the IB Trader Workstation connection `ib`.

```
close(ib)
```

#### See Also

`close` | `createOrder` | `getdata` | `history` | `ibtw`s | `timeseries`

#### Related Examples

- “Create Interactive Brokers Combination Order” on page 3-40
- “Create Interactive Brokers Order”
- “Request Interactive Brokers Real-Time Data”



## **More About**

- “Workflow for Interactive Brokers”

## **External Web Sites**

- Interactive Brokers API Reference Guide

## Request Interactive Brokers Real-Time Data

This example shows how to connect to the IB Trader Workstation, create IB Trader Workstation `IContract` objects, and request real-time data. For details about the `IContract` object, see *Interactive Brokers API Reference Guide*.

This example uses the sample event handler function `ibExampleRealtimeEventHandler` to handle events associated with requesting real-time data. Use this event handler or write a custom event handler function. For details, see “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20.

To access the code for this example, see `IBStreamingDataWorkflow.m`.

### Connect to the IB Trader Workstation and Create the Real-Time Data Display Figure

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

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

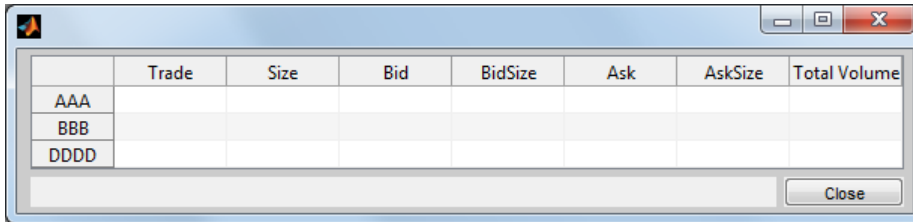
MATLAB returns `ib` as the connection to the IB Trader Workstation with the Interactive Brokers ActiveX object, the local host, and the port number that you choose.

Create an example figure to display real-time data.

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

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



	Trade	Size	Bid	BidSize	Ask	AskSize	Total Volume
AAA							
BBB							
DDDD							

### Create IB Trader Workstation IContract Objects

Create the IB Trader Workstation IContract object for the first security. Here, this object describes a security with these property values:

- AAA symbol
- Stock security type
- Aggregate exchange
- USD currency

```
ibContract1 = ib.Handle.createContract;
ibContract1.symbol = 'AAA';
ibContract1.secType = 'STK';
ibContract1.exchange = 'SMART';
ibContract1.currency = 'USD';
```

Create the IB Trader Workstation IContract object for the second security symbol BBB.

```
ibContract2 = ib.Handle.createContract;
ibContract2.symbol = 'BBB';
ibContract2.secType = 'STK';
ibContract2.exchange = 'SMART';
ibContract2.currency = 'USD';
```

Create the IB Trader Workstation IContract object for the third security symbol DDDD.

```
ibContract3 = ib.Handle.createContract;
ibContract3.symbol = 'DDDD';
ibContract3.secType = 'STK';
ibContract3.exchange = 'SMART';
ibContract3.currency = 'USD';
```

Display the data in the symbol property of ibContract1.

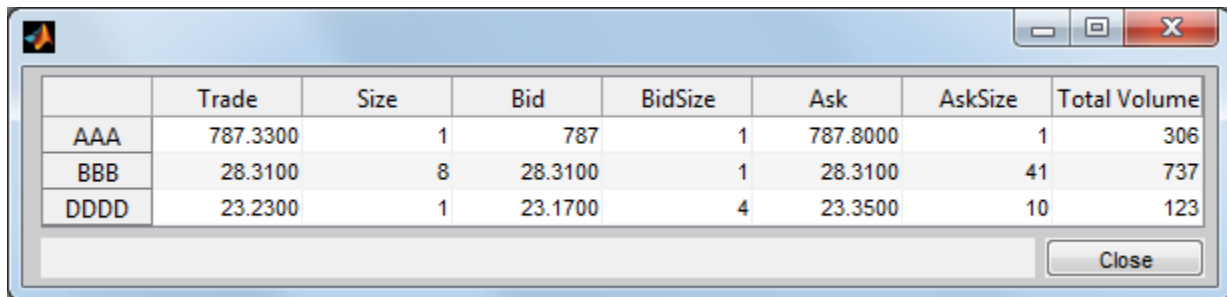
```
ibContract1.symbol
```

```
ans =  
    AAA
```

Request real-time data for the three securities. Set `f` to 100 to retrieve the Option Volume tick type. For details about other generic market data tick types, see Interactive Brokers API Reference Guide. Use the sample event handler `ibExampleRealtimeEventHandler` to process the real-time data events or write a custom event handler function.

```
contracts = {ibContract1;ibContract2;ibContract3};  
f = '100';  
  
tickerID = realtime(ib,contracts,f,...  
    @(varargin)ibExampleRealtimeEventHandler(varargin{:}));
```

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



The figure window displays a table with the following data:

	Trade	Size	Bid	BidSize	Ask	AskSize	Total Volume
AAA	787.3300	1	787	1	787.8000	1	306
BBB	28.3100	8	28.3100	1	28.3100	41	737
DDDD	23.2300	1	23.1700	4	23.3500	10	123

#### Close the Connection

Close the IB Trader Workstation connection `ib`.

```
close(ib)
```

#### See Also

`close` | `createOrder` | `getdata` | `history` | `ibtw` | `timeseries`

#### Related Examples

- “Create Interactive Brokers Combination Order” on page 3-40
- “Create Interactive Brokers Order”

- “Request Interactive Brokers Historical Data”

## **More About**

- “Workflow for Interactive Brokers”
- “Writing and Running Custom Event Handler Functions with Interactive Brokers”  
on page 1-20

## **External Web Sites**

- Interactive Brokers API Reference Guide

## Create Interactive Brokers Combination Order

This example shows how to connect to the IB Trader Workstation, create IB Trader Workstation `IContract` and `IComboLegList` objects, and create a combination order for a calendar spread. A calendar spread is one of many combination order strategies. This strategy takes advantage of different stock option expiration dates. This example creates a buy order on a calendar spread for Google<sup>®</sup>. For details about `IContract` objects, `IComboLegList` objects, and combination orders, see Interactive Brokers API Reference Guide.

This example uses the sample event handler function `ibExampleEventHandler` to handle events associated with creating a combination order. Use this event handler or write a custom event handler function. For details, see “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20.

To access the code for this example, see `IBCombinationOrder.m`.

### Connect to the IB Trader Workstation

Connect to the 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 port number that you choose.

### Create IB Trader Workstation `IContract` Objects

Create the IB Trader Workstation `IContract` object `ibContract1`. Here, this object describes the first call option in the calendar spread. Create an `IContract` object with these property values:

- Google symbol.
- Stock option.
- Expiry date is August 2014.
- Strike price is \$535.00.
- Call option.
- Number of shares is 100.
- Aggregate exchange.

- USD currency.

```
ibContract1 = ib.Handle.createContract;  
ibContract1.symbol = 'GOOG';  
ibContract1.secType = 'OPT';  
ibContract1.expiry = '201408';  
ibContract1.strike = 535;  
ibContract1.right = 'C';  
ibContract1.multiplier = '100';  
ibContract1.exchange = 'SMART';  
ibContract1.currency = 'USD';
```

Request contract details for `ibContract1`.

```
[cd1,ibContractID1] = contractdetails(ib,ibContract1);
```

`cd1` returns the contract details data for `ibContract1`. `ibContractID1` returns the request identifier for this contract details request.

Create the IB Trader Workstation `IContract` object `ibContract2`. Here, this object describes the second call option in the calendar spread. Create an `IContract` object with these property values:

- Google symbol.
- Stock option.
- Expiry date is September 2014.
- Strike price is \$535.00.
- Call option.
- Number of shares is 100.
- Aggregate exchange.
- USD currency.

```
ibContract2 = ib.Handle.createContract;  
ibContract2.symbol = 'GOOG';  
ibContract2.secType = 'OPT';  
ibContract2.expiry = '201409';  
ibContract2.strike = 535;  
ibContract2.right = 'C';  
ibContract2.multiplier = '100';  
ibContract2.exchange = 'SMART';  
ibContract2.currency = 'USD';
```

Request contract details for `ibContract2`.

```
[cd2,ibContractID2] = contractdetails(ib,ibContract2);
```

`cd2` returns the contract details data for `ibContract12`. `ibContractID2` returns the request identifier for this contract details request.

#### Create IB Trader Workstation `IComboLegList` Object

Create the IB Trader Workstation `IComboLegList` object `comboLegs` to define the legs of the combination order.

```
comboLegs = ib.Handle.createComboLegList;
```

Here, this combination order has two legs. Add the first leg to `comboLegs`. The first leg contains these property values:

- IB Trader Workstation `IContract` object `ibContract1`.
- One-to-one leg ratio.
- Sell the call option.
- Aggregate exchange.
- Identify an open or close order based on the parent security.
- IB Trader Workstation routes the order without a designated broker.
- Blank designated broker.

```
ibLeg1 = comboLegs.Add;  
ibLeg1.conId = ibContractID1;  
ibLeg1.ratio = 1;  
ibLeg1.action = 'SELL';  
ibLeg1.exchange = 'SMART';  
ibLeg1.openClose = 0;  
ibLeg1.shortSaleSlot = 0;  
ibLeg1.designatedLocation = '';
```

Add the second leg to `comboLegs`. The second leg contains these property values:

- IB Trader Workstation `IContract` object `ibContract2`.
- One-to-one leg ratio.
- Buy the call option.
- Aggregate exchange.



- Identify an open or close order based on the parent security.
- IB Trader Workstation routes the order without a designated broker.
- Blank designated broker.

```
ibLeg2 = comboLegs.Add;
ibLeg2.conId = ibContractID2;
ibLeg2.ratio = 1;
ibLeg2.action = 'BUY';
ibLeg2.exchange = 'SMART';
ibLeg2.openClose = 0;
ibLeg2.shortSaleSlot = 0;
ibLeg2.designatedLocation = '';
```

### Create the Interactive Brokers Combination Order

Create the IB Trader Workstation `IContract` object `orderContract` for the combination order. Create an `IContract` object with these property values:

- Google symbol
- Combination order type BAG
- Aggregate exchange
- USD currency
- IB Trader Workstation `IComboLegList` object `comboLegs`

```
orderContract = ib.Handle.createContract;
orderContract.symbol = 'GOOG';
orderContract.secType = 'BAG';
orderContract.exchange = 'SMART';
orderContract.currency = 'USD';
orderContract.comboLegs = comboLegs;
```

Create the IB Trader Workstation `IOrder` object `ibOrder`. Here, the combination order is a market order to buy one combination of the two legs.

```
ibOrder = ib.Handle.createOrder;
ibOrder.action = 'BUY';
ibOrder.totalQuantity = 1;
ibOrder.orderType = 'MKT';
```

Request the next valid order identification number `id` using `orderid`.

```
id = orderid(ib);
```

Execute the combination order `ibOrder` using these arguments:

- IB Trader Workstation connection `ib`
- Combination order `IContract` object `orderContract`
- IB Trader Workstation `IOrder` object `ibOrder`
- Order identifier `id`
- Sample event handler `ibExampleEventHandler`

```
d = createOrder(ib,orderContract,ibOrder,id,@ibExampleEventHandler)
```

```
d =
```

```
    768413.00
```

`d` returns the unique order identifier for this combination order.

### Close the Connection

Close the IB Trader Workstation connection `ib`.

```
close(ib)
```

### See Also

[close](#) | [contractdetails](#) | [createOrder](#) | [ibtws](#) | [orderid](#)

### Related Examples

- “Create Interactive Brokers Order”
- “Request Interactive Brokers Historical Data”
- “Request Interactive Brokers Real-Time Data”

### More About

- “Workflow for Interactive Brokers”
- “Writing and Running Custom Event Handler Functions with Interactive Brokers”  
on page 1-20

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

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

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

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

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

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

Establish the connection to CQG.

```
startUp(c)
```

```
CELStarted
DataConnectionStatusChanged
GWConnectionStatusChanged
```

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

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

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

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

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

#### Subscribe to the CQG Instrument

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

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

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

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

Create the CQG instrument object.

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

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

#### Set Up Account Credentials

Set the CQG flags to enable account information retrieval.

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

```
ans =
    AccountChanged
```

The CQG API shows that account information changed.

Set up the CQG account credentials.

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

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

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

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

```
quantity = 1;

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

ans =
    OrderChanged
```

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

To use a 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 details about the `CQGInstrument` object, see *CQG API Reference Guide*.

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

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

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

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

```
ans =
    OrderChanged
```

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

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

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

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

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

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

```
ans =
    OrderChanged
```

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

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

```
quantity = 1;

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

ans =
    OrderChanged
```

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

### Close the CQG Connection

```
shutDown(c)
```

### See Also

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

### Related Examples

- “Request CQG Historical Data”
- “Request CQG Real-Time Data”
- “Request CQG Intraday Tick Data”

### More About

- “Workflow for CQG”

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

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

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

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

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

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

Create the CQG connection.

```
startUp(c)
```

```
CELStarted  
DataConnectionStatusChanged
```

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

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



```

histEventNames = {'ExpressionResolved', 'ExpressionAdded', ...
    'ExpressionUpdated'};
for i = 1:length(histEventNames)
    registerevent(c.Handle, {histEventNames{i}, ...
        @(varargin)cqgexpressioneventhandler(varargin{:})})
end

```

### Pass an Additional Optional Request Property

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

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

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

### Request CQG Historical Data

Request daily data for instrument XYZ.XYZ for the last 10 days using the additional optional request property `x`.

```

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

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

### Related Examples

- “Create CQG Order”
- “Request CQG Real-Time Data”
- “Request CQG Intraday Tick Data”

### More About

- “Workflow for CQG”

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

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

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

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

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

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

Create the CQG connection.

```
startUp(c)
```

```
CELStarted
DataConnectionStatusChanged
```

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

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

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

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

#### Request CQG Intraday Tick Data

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

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

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

Request intraday tick data for instrument `XYZ.XYZ` for the last 2 days using the additional optional request property `x`.

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

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

### Close the CQG Connection

```
close(c)
```

### See Also

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

### Related Examples

- “Create CQG Order”
- “Request CQG Historical Data”
- “Request CQG Real-Time Data”

### More About

- “Workflow for CQG”

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

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

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

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

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

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

Establish the connection to CQG.

```
startUp(c)
```

```
CELStarted
DataConnectionStatusChanged
GWConnectionStatusChanged
```

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

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

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

#### Request CQG Real-Time Data

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

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

### Close the CQG Connection

```
close(c)
```

### See Also

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

### Related Examples

- “Create CQG Order”
- “Request CQG Historical Data”
- “Request CQG Intraday Tick Data”

### More About

- “Workflow for CQG”

### External Web Sites

- CQG API Reference Guide



# Functions — Alphabetical List

---

### emsx

Create Bloomberg EMSX connection

### Syntax

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

### Description

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

### Examples

#### Connect to the Bloomberg EMSX Test Service

Create a connection `c` to the Bloomberg EMSX test service. You can place test calls using this service.

```
c = emsx(' //blp/emapisvc_beta' )  
  
c =  
  
emsx with properties:  
  
    Session: [1x1 com.bloomberglp.blpapi.Session]  
    Service: [1x1 com.bloomberglp.blpapi.impl.aQ]  
    Ippaddress: 'localhost'  
    Port: 8194
```

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

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

- Port number of the machine running the Bloomberg EMSX test service

Close the Bloomberg EMSX connection.

```
close(c)
```

### Connect to the Bloomberg EMSX Production Service

Create a connection `c` to the Bloomberg EMSX production service. You can place live calls using this service.

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

```
c =
```

```
emsx with properties:
```

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## 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** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, returned as a connection object with these properties.

Property	Description
Session	Bloomberg EMSX session object
Service	Bloomberg EMSX service object
Ippaddress	IP address of the machine where Bloomberg EMSX is running
Port	Port number of the machine where Bloomberg EMSX is running

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”

### See Also

close | createOrder | createOrderAndRoute | routeOrder

## close

Close Bloomberg EMSX connection

## Syntax

```
close(c)
```

## Description

`close(c)` closes the Bloomberg EMSX connection `c`.

## Examples

### Close the Bloomberg EMSX Connection

Create the Bloomberg EMSX connection `c`.

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

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”

### See Also

`createOrder` | `createOrderAndRoute` | `emsx` | `routeOrder`



# createOrder

Create Bloomberg EMSX order

## Syntax

```
events = createOrder(c,order)
events = createOrder(c,order,'timeOut',timeout)

createOrder( ____, 'useDefaultEventHandler',false)

____ = createOrder(,c,order,options)
```

## Description

`events = createOrder(c,order)` creates a Bloomberg EMSX order using the Bloomberg EMSX connection `c` and order request `order` that contains the required fields for creating an order. `createOrder` returns the order sequence number and status message using the default event handler.

`events = createOrder(c,order,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`createOrder( ____, 'useDefaultEventHandler',false)` creates a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with creating orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`____ = createOrder(,c,order,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

## Examples

### Create an Order Using the Default Event Handler

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”.

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

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

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

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

```
events =
```

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

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

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Create an Order Using a Timeout

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”.

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

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

Create the order using the Bloomberg EMSX connection `c` and `order`. Set the timeout value to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code.

```
events = createOrder(c,order,'timeOut',200)
```

```
events =
```

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

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

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Create an Order Using a Custom Event Handler

To create a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”.

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

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

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17.

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

`t` is the MATLAB timer object. For details, see `timer`.

Create the order using the Bloomberg EMSX connection `c` and `order`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with creating an order.

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

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`. Stop the timer to stop data updates using `stop`.

```
b.Session.unsubscribe(subs)
```

```
stop(t)
```

Delete the timer if you are done processing data updates using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Create an Order Using an Options Structure

To create a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order subscription using `orders`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”.

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

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

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code. Create the order using the Bloomberg EMSX connection `c`, `order`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;
```

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

```
events =
```

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

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

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

### **order** — Order request

structure

Order request, specified as a structure using Bloomberg EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `order`. Convert the number of shares to a 32-bit signed integer using `int32`. `order` contains these fields.

Field	Description
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX amount of shares
EMSX_ORDER_TYPE	Bloomberg EMSX order type
EMSX_BROKER	Bloomberg EMSX broker name

Field	Description
EMSX_TIF	Bloomberg EMSX time in force
EMSX_HAND_INSTRUCTION	Bloomberg EMSX hand instruction
EMSX_SIDE	Bloomberg EMSX buy or sell specification

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

Data Types: struct

#### **timeout** — Timeout value

500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer to denote the time in milliseconds the default or custom event handler waits for an event in the current queue.

Data Types: double

#### **options** — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure to reuse the settings for specifying a custom event handler or timeout value for the event handler.

Specify using a custom event handler and a timeout value of 500 milliseconds.

```
Example: options.useDefaultEventHandler = false;
options.timeOut = 500;
```

Data Types: struct

## Output Arguments

#### **events** — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”
- “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17

### See Also

`timer` | `close` | `createOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `delete` | `deleteOrder` | `deleteRoute` | `emx` | `modifyOrder` | `orders` | `routeOrder` | `routes` | `start` | `stop`



# createOrderAndRoute

Create and route Bloomberg EMSX order

## Syntax

```
events = createOrderAndRoute(c,order)
events = createOrderAndRoute(c,order,'timeOut',timeout)
createOrderAndRoute( ____, 'useDefaultEventHandler',false)
____ = createOrderAndRoute(c,order,options)
```

## Description

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

`events = createOrderAndRoute(c,order,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`createOrderAndRoute( ____, 'useDefaultEventHandler',false)` creates and routes a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with creating and routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`____ = createOrderAndRoute(c,order,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

## Examples

### Create and Route an Order Using the Default Event Handler

To create and route a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route”.

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

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

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

```
events = createOrderAndRoute(c,order)

events =

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

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

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

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

```
b.Session.unsubscribe(osubs)
```

```
b.Session.unsubscribe(rsubs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Create and Route an Order Using a Timeout

To create and route a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route”.

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

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

Create and route the order using the Bloomberg EMSX connection `c` and `order`. Set the timeout value to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code.

```
events = createOrderAndRoute(c,order,'timeOut',200)
```

```
events =
```

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

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

### Create and Route an Order Using a Custom Event Handler

To create and route a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route”.

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

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

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17.

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

`t` is the MATLAB timer object. For details, see `timer`.

Create and route the order using the Bloomberg EMSX connection `c` and `order`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with creating and routing an order.

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

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

```
b.Session.unsubscribe(osubs)
b.Session.unsubscribe(rsubs)
stop(t)
```

Delete the timer if you are done processing data updates using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Create and Route an Order Using an Options Structure

To create and route a Bloomberg EMSX order, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route”.

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

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

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code. Create and route the order using the Bloomberg EMSX connection `c`, `order`, and options structure `options`.

```
options.useDefaultEventHandler = true;
```

```
options.timeOut = 200;

events = createOrderAndRoute(c,order,options)

events =
    EMSX_SEQUENCE: 728924
    EMSX_ROUTE_ID: 1
    MESSAGE: 'Order created and routed'
```

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

### **order** — Order request

structure

Order request, specified as a structure using Bloomberg EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `order`. Convert the number of shares to a 32-bit signed integer using `int32`. `order` contains these fields.

Field	Description
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX amount of shares
EMSX_ORDER_TYPE	Bloomberg EMSX order type
EMSX_BROKER	Bloomberg EMSX broker name
EMSX_TIF	Bloomberg EMSX time in force
EMSX_HAND_INSTRUCTION	Bloomberg EMSX hand instruction
EMSX_SIDE	Bloomberg EMSX buy or sell specification

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

Data Types: struct

#### **timeout** — Timeout value

500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer to denote the time in milliseconds the default or custom event handler waits for an event in the current queue.

Data Types: double

#### **options** — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure to reuse the settings for specifying a custom event handler or timeout value for the event handler.

Specify using a custom event handler and a timeout value of 500 milliseconds.

```
Example: options.useDefaultEventHandler = false;  
options.timeOut = 500;
```

Data Types: struct

## Output Arguments

### **events** — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, **events** is a structure containing the current contents of the event queue. Otherwise, **events** is an empty double.

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”
- “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17

### See Also

timer | close | createOrder | createOrderAndRouteWithStrat | delete | deleteOrder | deleteRoute | emsx | modifyOrder | orders | routeOrder | routes | start | stop



# createOrderAndRouteWithStrat

Create and route Bloomberg EMSX order with strategies

## Syntax

```
events = createOrderAndRouteWithStrat(c,order,strat)
events = createOrderAndRouteWithStrat(c,order,strat,'timeOut',
timeout)

createOrderAndRouteWithStrat( ____, 'useDefaultEventHandler',false)

____ = createOrderAndRouteWithStrat(c,order,strat,options)
```

## Description

`events = createOrderAndRouteWithStrat(c,order,strat)` creates and routes a Bloomberg EMSX order with strategies using Bloomberg EMSX connection `c`, order request `order`, and order strategy `strat`. `createOrderAndRouteWithStrat` returns the order sequence number, route number, and status message using the default event handler.

`events = createOrderAndRouteWithStrat(c,order,strat,'timeOut', timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`createOrderAndRouteWithStrat( ____, 'useDefaultEventHandler',false)` creates and routes a Bloomberg EMSX order with strategies using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with creating and routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`____ = createOrderAndRouteWithStrat(c,order,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are

`timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

## Examples

### Create and Route an Order Using the Default Event Handler

To create and route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route”.

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

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

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

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

Create and route the order with strategies using the Bloomberg EMSX connection `c`, `order`, and `strat`.

```
events = createOrderAndRouteWithStrat(c,order,strat)
```

```
events =
```

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

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

### Create and Route an Order Using a Timeout

To create and route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route”.

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

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

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMSX_STRATEGY_NAME = 'SSP';
```

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

Create and route the order with strategies using the Bloomberg EMSX connection `c`, `order`, and `strat`. Set the timeout value to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code.

```
events = createOrderAndRouteWithStrat(c,order,strat,'timeOut',200)
```

```
events =
```

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

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

### Create and Route an Order Using a Custom Event Handler

To create and route a Bloomberg EMSX order with strategies, create the Bloomberg EMSX connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route”.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 100 shares of IBM. This code uses the broker `BB` with the

time in force set to DAY and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

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

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

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

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17.

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

`t` is the MATLAB timer object. For details, see `timer`.

Create and route the order with strategies using the Bloomberg EMSX connection `c`, `order`, and `strat`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with creating and routing an order.

```
createOrderAndRouteWithStrat(c,order,strat,...
                             'useDefaultEventHandler',false)
```

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

```
b.Session.unsubscribe(osubs)
b.Session.unsubscribe(rsubs)
stop(t)
```

Delete the timer if you are done processing data updates using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Create and Route an Order Using an Options Structure

To create and route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx` and set up the order and route subscription using `orders` and `routes`. For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route”.

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

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

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

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

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code. Create and route the order using the Bloomberg EMSX connection `c`, `order`, `strat`, and `options` structure `options`.

```
options.useDefaultEventHandler = true;  
options.timeOut = 200;
```

```
events = createOrderAndRouteWithStrat(c,order,strat,options)
```

```
events =
```

```
  EMSX_SEQUENCE: 728924
  EMSX_ROUTE_ID: 1
  MESSAGE: 'Order created and routed'
```

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

### **order** — Order request

structure

Order request, specified as a structure using Bloomberg EMSX field properties. Use `getAllFieldMetaData` to view all available field property options for `order`. Convert the number of shares to a 32-bit signed integer using `int32`. `order` contains these fields.

Field	Description
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX amount of shares
EMSX_ORDER_TYPE	Bloomberg EMSX order type
EMSX_BROKER	Bloomberg EMSX broker name
EMSX_TIF	Bloomberg EMSX time in force
EMSX_HAND_INSTRUCTION	Bloomberg EMSX hand instruction
EMSX_SIDE	Bloomberg EMSX buy or sell specification

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

Data Types: struct

### **strat** — Order strategies

structure

Order strategies, specified as a structure that contains the fields: `EMSX_STRATEGY_NAME`, `EMSX_STRATEGY_FIELD_INDICATORS`, and `EMSX_STRATEGY_FIELDS`. The structure field values must align with the strategy fields specified by `EMSX_STRATEGY_NAME`. For details about strategy fields and ordering, see `getBrokerInfo`.

Convert `EMSX_STRATEGY_FIELD_INDICATORS` to a 32-bit signed integer using `int32`. Set `EMSX_STRATEGY_FIELD_INDICATORS` equal to 0 for each field to use the field data setting in `EMSX_FIELD_DATA`. Or, set `EMSX_STRATEGY_FIELD_INDICATORS` equal to 1 to ignore the data in `EMSX_FIELD_DATA`.

```
Example: strat.EMSX_STRATEGY_NAME = 'SSP';
strat.EMSX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
```



```
strat.EMSX_STRATEGY_FIELDS = {'09:30:00', '14:30:00', 50};
```

Data Types: `struct`

#### **timeout** — Timeout value

500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer to denote the time in milliseconds the default or custom event handler waits for an event in the current queue.

Data Types: `double`

#### **options** — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure to reuse the settings for specifying a custom event handler or timeout value for the event handler.

Specify using a custom event handler and a timeout value of 500 milliseconds.

```
Example: options.useDefaultEventHandler = false;
options.timeOut = 500;
```

Data Types: `struct`

## Output Arguments

#### **events** — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, **events** is a structure containing the current contents of the event queue. Otherwise, **events** is an empty double.

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”
- “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17

### **See Also**

timer | close | createOrder | delete | deleteOrder | deleteRoute | emsx | getBrokerInfo | modifyOrder | orders | routeOrder | routes | start | stop

# deleteOrder

Delete Bloomberg EMSX order

## Syntax

```
events = deleteOrder(c,ordernum)
events = deleteOrder(c,ordernum,'timeOut',timeout)

deleteOrder( ____, 'useDefaultEventHandler',false)

____ = deleteOrder(c,ordernum,options)
```

## Description

`events = deleteOrder(c,ordernum)` deletes a Bloomberg EMSX order using the Bloomberg EMSX connection `c` and order number or structure `ordernum`. `deleteOrder` returns a status message using the default event handler.

`events = deleteOrder(c,ordernum,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`deleteOrder( ____, 'useDefaultEventHandler',false)` deletes a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with deleting orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`____ = deleteOrder(c,ordernum,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

## Examples

### Delete an Order Using the Default Event Handler

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Delete the order using the Bloomberg EMSX connection `c` and `ordernum`.

```
events = deleteOrder(c,ordernum)
```

```
events =
```

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

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

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Delete an Order Using the Order Number Integer

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”.

Delete the order using the Bloomberg EMSX connection `c` and the order sequence number `335877` for the order to delete.

```
events = deleteOrder(c,335877)

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

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

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Delete an Order Using a Timeout

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Delete the order using the Bloomberg EMSX connection `c` and `ordernum`. Set the timeout value to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code.

```
events = deleteOrder(c,ordernum,'timeOut',200)

events =
```

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

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

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Delete an Order Using a Custom Event Handler

To delete a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17.

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

`t` is the MATLAB timer object. For details, see `timer`.

Delete the order using the Bloomberg EMSX connection `c` and `ordernum`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with deleting an order.

```
deleteOrder(c,ordernum, 'useDefaultEventHandler', false)
```

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`. Stop the timer to stop data updates using `stop`.

```
b.Session.unsubscribe(subs)
stop(t)
```

Delete the timer if you are done processing data updates using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Delete an Order Using an Options Structure

To delete a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”.

Define the structure `ordernum` that contains the order sequence number `EMSX_SEQUENCE` for the order to delete.

```
ordernum.EMSX_SEQUENCE = 335877;
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code. Delete the order using the Bloomberg EMSX connection `c`, `ordernum`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;
```

```
events = deleteOrder(c,ordernum,options)
```

```
events =
```

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

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

- Bloomberg EMSX status
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes `orders` creates `subs`.

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

### **ordernum** — Order numbers to delete

structure | integer

Order numbers to delete, specified as a structure or an integer to denote one or more order sequence numbers.

Data Types: `struct` | `int32`

### **timeout** — Timeout value

500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer to denote the time in milliseconds the default or custom event handler waits for an event in the current queue.

Data Types: `double`

### **options** — Options for custom event handler or timeout value

structure



Options for custom event handler or timeout value, specified as a structure. Use the options structure to reuse the settings for specifying a custom event handler or timeout value for the event handler.

Specify using a custom event handler and a timeout value of 500 milliseconds.

```
Example: options.useDefaultEventHandler = false;  
options.timeOut = 500;
```

Data Types: struct

## Output Arguments

### **events** — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, **events** is a structure containing the current contents of the event queue. Otherwise, **events** is an empty double.

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”
- “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17

### See Also

timer | close | createOrder | createOrderAndRoute | delete | deleteRoute | emsx | modifyOrder | orders | routeOrder | routes | start | stop

## deleteRoute

Delete Bloomberg EMSX active shares

### Syntax

```
events = deleteRoute(c,routenum)
events = deleteRoute(c,routenum,'timeOut',timeout)
```

```
deleteRoute( ____, 'useDefaultEventHandler', false)
```

```
____ = deleteRoute(c,routenum,options)
```

### Description

`events = deleteRoute(c,routenum)` deletes the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and route number `routenum`. `deleteRoute` returns a status message using the default event handler.

`events = deleteRoute(c,routenum,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`deleteRoute( ____, 'useDefaultEventHandler', false)` deletes the active shares that are routed but not filled using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with deleting the active shares. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`____ = deleteRoute(c,routenum,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

## Examples

### Delete Active Shares

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

- 1 Create the connection `c` using `emsx`.
- 2 Set up an order and route subscription using `orders` and `routes`.
- 3 Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see “Create and Manage a Bloomberg EMSX Route”.

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

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and `routenum`.

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

```
events =
```

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

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

- Bloomberg EMSX status
- Bloomberg EMSX message

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

### Delete Active Shares Using a Timeout

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

- 1 Create the connection `c` using `emsx`.
- 2 Set up an order and route subscription using `orders` and `routes`.
- 3 Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see “Create and Manage a Bloomberg EMSX Route”.

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

```
routenum.EMSX_SEQUENCE = 335877;  
routenum.EMSX_ROUTE_ID = 1;
```

Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and `routenum`. Set the timeout value to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code.

```
options.useDefaultEventHandler = true;  
options.timeOut = 200;  
  
events = deleteRoute(c,routenum,'timeOut',200)  
  
events =
```

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

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

- Bloomberg EMSX status
- Bloomberg EMSX message

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

### Delete Active Shares Using a Custom Event Handler

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

- 1 Create the Bloomberg EMSX connection `c` using `emsx`.
- 2 Set up an order and route subscription using `orders` and `routes`.
- 3 Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see “Create and Manage a Bloomberg EMSX Route”.

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

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17.

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

`t` is the MATLAB timer object. For details, see `timer`.

Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c` and `routenum`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with deleting the active shares.

```
deleteRoute(c,routenum,'useDefaultEventHandler',false)
```

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

```
b.Session.unsubscribe(osubs)
b.Session.unsubscribe(rsubs)
stop(t)
```

Delete the timer if you are done processing data updates using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Delete Active Shares Using an Options Structure

To delete the active shares that are routed but not filled for a Bloomberg EMSX order:

- 1 Create the connection `c` using `emsx`.
- 2 Set up an order and route subscription using `orders` and `routes`.
- 3 Create and route an order using `createOrderAndRoute`.

For an example showing these activities, see “Create and Manage a Bloomberg EMSX Route”.

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

```
routenum.EMSX_SEQUENCE = 335877;
routenum.EMSX_ROUTE_ID = 1;
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code. Delete the active shares that are routed but not filled using the Bloomberg EMSX connection `c`, `routenum`, and `options` structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = deleteRoute(c,routenum,options)

events =

    STATUS: '1'
```

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

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

- Bloomberg EMSX status
- Bloomberg EMSX message

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

### **routenum** — Route to delete

structure

Route to delete, specified as a structure containing fields `EMSX_SEQUENCE` and `EMSX_ROUTE_ID`.

```
Example: routenum.EMSX_SEQUENCE = 728918;
routenum.EMSX_ROUTE_ID = 1;
```

Data Types: struct

### **timeout** — Timeout value

500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer to denote the time in milliseconds the default or custom event handler waits for an event in the current queue.

Data Types: `double`

### **options** — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure to reuse the settings for specifying a custom event handler or timeout value for the event handler.

Specify using a custom event handler and a timeout value of 500 milliseconds.

```
Example: options.useDefaultEventHandler = false;  
options.timeOut = 500;
```

Data Types: `struct`

## Output Arguments

### **events** — Event queue contents

`double` | structure

Event queue contents, returned as a `double` or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty `double`.

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”



- “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17

**See Also**

timer | close | createOrder | createOrderAndRoute | delete | deleteOrder  
| emsx | modifyOrder | modifyRoute | orders | routeOrder | routes | start |  
stop

## getAllFieldMetaData

Obtain Bloomberg EMSX field information

### Syntax

```
r = getAllFieldMetaData(c)
```

### Description

`r = getAllFieldMetaData(c)` returns the Bloomberg EMSX field information using the Bloomberg EMSX connection `c`.

### Examples

#### Request All Field Information

Create a connection `c` to the Bloomberg EMSX.

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

Request all fields supported by Bloomberg EMSX service using the Bloomberg EMSX connection `c`.

```
r = getAllFieldMetaData(c)
```

```
r =
```

```
    EMSX_FIELD_NAME: {113x1 cell}  
    EMSX_DISP_NAME: {113x1 cell}  
    EMSX_TYPE: {113x1 cell}  
    EMSX_LEVEL: [113x1 double]  
    EMSX_LEN: [113x1 double]
```

Display all field information for the first Bloomberg EMSX field using a cell array. Create a cell array from the fields in the returned data structure `r`.

```
{r.EMMX_FIELD_NAME{1} r.EMMX_DISP_NAME{1} r.EMMX_TYPE{1} r.EMMX_LEVEL(1) r.EMMX_LEN(1)}
```

```
'MSG_TYPE'      'Msg Type'      'String'      [0]      [1]
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

**c** — **Bloomberg EMSX service connection**

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

## Output Arguments

**r** — **Return information for all fields**

structure

Return information for all fields, returned as a structure for all fields supported by Bloomberg EMSX.

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”

### See Also

```
close | createOrder | createOrderAndRoute |
createOrderAndRouteWithStrat | emsx
```

# getBrokerInfo

Obtain Bloomberg EMSX broker and strategy information

## Syntax

```
r = getBrokerInfo(c,brokerstrat)
```

## Description

`r = getBrokerInfo(c,brokerstrat)` obtains Bloomberg EMSX broker and strategy information using the Bloomberg EMSX connection `c` and broker and strategy request structure `brokerstrat`.

## Examples

### Obtain Broker Information

Create a connection `c` to the Bloomberg EMSX.

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

Define the broker and strategy information structure `brokerstrat`. Obtain broker information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
```

```
r = getBrokerInfo(c,brokerstrat)
```

```
r =
```

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

The `EMSX_BROKERS` field lists the Bloomberg EMSX brokers.

Close the Bloomberg EMSX connection.

```
close(c)
```

### Obtain Strategy Information

Create a connection `c` to the Bloomberg EMSX.

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

Define the broker and strategy information structure `brokerstrat`. Obtain strategy information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
brokerstrat.EMSX_BROKER = 'BMTB';
```

```
r = getBrokerInfo(c,brokerstrat)
```

```
r =
```

```
    EMSX_STRATEGIES: {16x1 cell}
```

The `EMSX_STRATEGIES` field lists the Bloomberg EMSX strategies.

Close the Bloomberg EMSX connection.

```
close(c)
```

### Obtain Field Information

Create a connection `c` to the Bloomberg EMSX.

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

Define the broker and strategy information structure `brokerstrat`. Obtain field information using the Bloomberg EMSX connection `c` and structure `brokerstrat`.

```
brokerstrat.EMSX_TICKER = 'ABCD US Equity';
brokerstrat.EMSX_BROKER = 'BMTB';
brokerstrat.EMSX_STRATEGY = 'SSP';
```

```
r = getBrokerInfo(c,brokerstrat)
```

```
r =
```

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

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

The structure field `FieldName` lists the Bloomberg EMSX fields. The structure fields `Disable` and `StringValue` contain information about the Bloomberg EMSX fields.

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

### **brokerstrat** — Broker and strategy request

structure

Broker and strategy request, specified as a structure that contains Bloomberg EMSX fields. Use `getAllFieldMetaData` to view all available fields for `brokerStrategyStruct`.

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

Data Types: struct

## Output Arguments

### **r** — Broker and strategy information

structure

Broker and strategy information, returned as a structure.

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”

### See Also

`close` | `createOrder` | `createOrderAndRoute` |  
`createOrderAndRouteWithStrat` | `deleteOrder` | `deleteRoute` | `emsx` |  
`modifyOrder` | `orders` | `routeOrder` | `routes`

## modifyOrder

Modify Bloomberg EMSX order

### Syntax

```
events = modifyOrder(c,modorder)
events = modifyOrder(c,modorder,'timeOut',timeout)
```

```
modifyOrder( ____, 'useDefaultEventHandler', false)
```

```
____ = modifyOrder(c,modorder,options)
```

### Description

`events = modifyOrder(c,modorder)` modifies a Bloomberg EMSX order using the Bloomberg EMSX connection `c` and modify order request structure `modorder`. `modifyOrder` returns a status message using the default event handler.

`events = modifyOrder(c,modorder,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`modifyOrder( ____, 'useDefaultEventHandler', false)` modifies a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with modifying orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`____ = modifyOrder(c,modorder,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.



## Examples

### Modify an Order Using the Default Event Handler

To modify a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number `728905` for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

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

```
events = modifyOrder(c,modorder)

events =
    EMSX_SEQUENCE: 728905
    MESSAGE: 'Order Modified'
```

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

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Modify an Order Using a Timeout

To modify a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number 728905 for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Modify the order using the Bloomberg EMSX connection `c` and `modorder`. Set the timeout value to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code.

```
events = modifyOrder(c,modorder,'timeOut',200)
```

```
events =
```

```
    EMSX_SEQUENCE: 728905
      MESSAGE: 'Order Modified'
```

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

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Modify an Order Using a Custom Event Handler

To modify a Bloomberg EMSX order, create the Bloomberg EMSX connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number 728905 for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17.

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

`t` is the MATLAB timer object. For details, see `timer`.

Modify the order using the Bloomberg EMSX connection `c` and `modorder`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with modifying an order.

```
modifyOrder(c,modorder,'useDefaultEventHandler',false)
```

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`. Stop the timer to stop data updates using `stop`.

```
b.Session.unsubscribe(subs)
stop(t)
```

Delete the timer if you are done processing data updates using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Modify an Order Using an Options Structure

To modify a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create an order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”.

Define the structure `modorder` that contains the order sequence number `EMSX_SEQUENCE`, the security `EMSX_TICKER`, and the number of shares `EMSX_AMOUNT`. This code modifies the order number 728905 for 200 shares of IBM. Convert the numbers to 32-bit signed integers using `int32`.

```
modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'IBM';
modorder.EMSX_AMOUNT = int32(200);
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code. Modify the order using the Bloomberg EMSX connection `c`, `modorder`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = modifyOrder(c,modorder,options)

events =
    EMSX_SEQUENCE: 728905
    MESSAGE: 'Order Modified'
```

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

- Bloomberg EMSX order number
- Bloomberg EMSX message

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. This code assumes that `orders` creates `subs`.

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

### **modorder** — Modify order request

structure

Modify order request, specified as a structure that contains these fields.

Use `getAllFieldMetaData` to view all available fields for `modorder`. Convert the numbers to 32-bit signed integers using `int32`.

Field	Description
EMSX_SEQUENCE	Bloomberg EMSX order sequence number
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX number of shares

```
Example: modorder.EMSX_SEQUENCE = int32(728905);
modorder.EMSX_TICKER = 'XYZ';
modorder.EMSX_AMOUNT = int32(100);
```

Data Types: struct

### **timeout** — Timeout value

500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer to denote the time in milliseconds the default or custom event handler waits for an event in the current queue.

Data Types: double

### **options** — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure to reuse the settings for specifying a custom event handler or timeout value for the event handler.

Specify using a custom event handler and a timeout value of 500 milliseconds.

```
Example: options.useDefaultEventHandler = false;  
options.timeOut = 500;
```

Data Types: struct

## Output Arguments

### **events** — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, **events** is a structure containing the current contents of the event queue. Otherwise, **events** is an empty double.

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”

- “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17

**See Also**

timer | close | createOrder | createOrderAndRoute |  
createOrderAndRouteWithStrat | delete | deleteOrder | deleteRoute | emsx  
| orders | routeOrder | routes | start | stop

## modifyRoute

Modify Bloomberg EMSX route

### Syntax

```
events = modifyRoute(c,modroute)
events = modifyRoute(c,modroute,'timeOut',timeout)
```

```
modifyRoute( ____, 'useDefaultEventHandler', false)
```

```
____ = modifyRoute(c,modroute,options)
```

### Description

`events = modifyRoute(c,modroute)` modifies a Bloomberg EMSX route using the Bloomberg EMSX connection `c` and route request `modroute`. `modifyRoute` returns a status message using the default event handler.

`events = modifyRoute(c,modroute,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`modifyRoute( ____, 'useDefaultEventHandler', false)` modifies a Bloomberg EMSX route using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with modifying routes. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`____ = modifyRoute(c,modroute,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.



## Examples

### Modify a Route Using the Default Event Handler

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route” on page 3-22.

Define the `modroute` structure that contains these fields:

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

This code instructs Bloomberg EMSX to route 100 shares of IBM for order sequence number 731128 and route identifier 1. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

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

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

```
events =
```

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

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

### Modify a Route Using a Timeout

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route” on page 3-22.

Define the `modroute` structure that contains these fields:

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

This code modifies the route to 100 shares of IBM for order sequence number 731128 and route identifier 1. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Modify the route using the Bloomberg EMSX connection `c` and `modroute`. Set the timeout value to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code.

```
events = modifyRoute(c,modroute, 'timeOut',200)

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

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

### Modify a Route Using a Custom Event Handler

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route” on page 3-22.

Define the `modroute` structure that contains these fields:

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

This code modifies the route to 100 shares of IBM for order sequence number 731128 and route identifier 1. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17.

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

`t` is the MATLAB timer object. For details, see `timer`.

Modify the route using the Bloomberg EMSX connection `c` and `modroute`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with modifying a route.

```
modifyRoute(c,modroute, 'useDefaultEventHandler',false)
```

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

```
b.Session.unsubscribe(osubs)
b.Session.unsubscribe(rsubs)
stop(t)
```

Delete the timer if you are done processing data updates using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Modify a Route Using an Options Structure

To modify a route for a Bloomberg EMSX order:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route” on page 3-22.

Define the `modroute` structure that contains these fields:

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

This code modifies the route to 100 shares of IBM for order sequence number 731128 and route identifier 1. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code. Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and `options` structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = modifyRoute(c,modroute,options)

events =
```

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

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

### **modroute** — Modify route request

structure

Modify route request, specified as a structure with these fields.

Use `getAllFieldMetaData` to view all available fields for `modroute`. Convert the numbers to 32-bit signed integers using `int32`.

Field	Description
EMSX_SEQUENCE	Bloomberg EMSX order sequence number
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX number of shares
EMSX_ROUTE_ID	Bloomberg EMSX route identifier

```
Example: modroute.EMSX_SEQUENCE = int32(731128);
modroute.EMSX_TICKER = 'XYZ';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Data Types: struct

#### **timeout** — Timeout value

500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer to denote the time in milliseconds the default or custom event handler waits for an event in the current queue.

Data Types: double

#### **options** — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure to reuse the settings for specifying a custom event handler or timeout value for the event handler.

Specify using a custom event handler and a timeout value of 500 milliseconds.

```
Example: options.useDefaultEventHandler = false;
options.timeOut = 500;
```

Data Types: struct

## Output Arguments

#### **events** — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”
- “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17

### See Also

`timer` | `createOrder` | `createOrderAndRoute` | `delete` | `deleteOrder` | `modifyRouteWithStrat` | `orders` | `routes` | `start` | `stop`



# modifyRouteWithStrat

Modify route with strategies for Bloomberg EMSX

## Syntax

```
events = modifyRouteWithStrat(c,modroute,strat)
events = modifyRouteWithStrat(c,modroute,strat,'timeOut',timeout)

modifyRouteWithStrat( ____, 'useDefaultEventHandler', false)

____ = modifyRouteWithStrat(c,modroute,strat,options)
```

## Description

`events = modifyRouteWithStrat(c,modroute,strat)` modifies a Bloomberg EMSX route with strategies using the Bloomberg EMSX connection `c`, route request `modroute`, and order strategy `strat`. `modifyRouteWithStrat` returns the order sequence number, route identifier, and status message using the default event handler.

`events = modifyRouteWithStrat(c,modroute,strat,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`modifyRouteWithStrat( ____, 'useDefaultEventHandler', false)` modifies a Bloomberg EMSX route with strategies using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with modifying routes. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`____ = modifyRouteWithStrat(c,modroute,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

## Examples

### Modify a Route with Strategies Using the Default Event Handler

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route” on page 3-22.

Define the `modroute` structure that contains these fields:

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

This code modifies the route to 100 shares of IBM for order sequence number 731128 and route identifier 1. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

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

Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and `strat`.

```
events = modifyRouteWithStrat(c,modroute,strat)

events =

    EMSX_SEQUENCE: 0
    EMSX_ROUTE_ID: 0
```

```
MESSAGE: 'Route modified'
```

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

### **Modify a Route with Strategies Using a Timeout**

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route” on page 3-22.

Define the `modroute` structure that contains these fields:

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

This code modifies the route to 100 shares of IBM for order sequence number 731128 and route identifier 1. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
```

```
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

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

Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and `strat`. Set the timeout value to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code.

```
events = modifyRouteWithStrat(c, modroute, strat, 'timeOut', 200)
```

```
events =
```

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

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

### **Modify a Route with Strategies Using a Custom Event Handler**

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route” on page 3-22.

Define the `modroute` structure that contains these fields:

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

This code modifies the route to 100 shares of IBM for order sequence number 731128 and route identifier 1. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

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

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17.

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

`t` is the MATLAB timer object. For details, see `timer`.

Modify the route using the Bloomberg EMSX connection `c`, `modroute`, and `strat`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with modifying a route.

```
modifyRouteWithStrat(c,modroute, strat, 'useDefaultEventHandler',false)
```

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

```
b.Session.unsubscribe(osubs)
b.Session.unsubscribe(rsubs)
stop(t)
```

Delete the timer if you are done processing data updates using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Modify a Route with Strategies Using an Options Structure

To modify a route for a Bloomberg EMSX order with strategies:

- Create the connection `c` using `emsx`.
- Set up the order and route subscription using `orders` and `routes`.
- Create and route the order using `createOrderAndRoute`.

For an example showing these activities, see “Manage a Bloomberg EMSX Order and Route” on page 3-22.

Define the `modroute` structure that contains these fields:

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

This code modifies the route to 100 shares of IBM for order sequence number 731128 and route identifier 1. Convert the numbers to 32-bit signed integers using `int32`.

```
modroute.EMSX_SEQUENCE = int32(731128)
modroute.EMSX_TICKER = 'IBM';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

```
strat.EMX_STRATEGY_NAME = 'SSP';
strat.EMX_STRATEGY_FIELD_INDICATORS = int32([0 0 0]);
strat.EMX_STRATEGY_FIELDS = {'09:30:00', '14:30:00', 50};
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code. Modify the route using the Bloomberg EMSX connection `c`, `modroute`, `strat`, and `options` structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;

events = modifyRouteWithStrat(c, modroute, strat, options)

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

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”

- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

### **modroute** — Modify route request

structure

Modify route request, specified as a structure with these fields.

Use `getAllFieldMetaData` to view all available fields for `modroute`. Convert the numbers to 32-bit signed integers using `int32`.

Field	Description
EMSX_SEQUENCE	Bloomberg EMSX order sequence number
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX number of shares
EMSX_ROUTE_ID	Bloomberg EMSX route identifier

```
Example: modroute.EMSX_SEQUENCE = int32(731128);
modroute.EMSX_TICKER = 'XYZ';
modroute.EMSX_AMOUNT = int32(100);
modroute.EMSX_ROUTE_ID = int32(1);
```

Data Types: struct

### **strat** — Order strategies

structure

Order strategies, specified as a structure that contains the fields:

`EMSX_STRATEGY_NAME`, `EMSX_STRATEGY_FIELD_INDICATORS`, and `EMSX_STRATEGY_FIELDS`. The structure field values must align with the strategy fields specified by `EMSX_STRATEGY_NAME`. For details about strategy fields and ordering, see `getBrokerInfo`.



Convert `EMSX_STRATEGY_FIELD_INDICATORS` to a 32-bit signed integer using `int32`. Set `EMSX_STRATEGY_FIELD_INDICATORS` equal to 0 for each field to use the field data setting in `EMSX_FIELD_DATA`. Or, set `EMSX_STRATEGY_FIELD_INDICATORS` equal to 1 to ignore the data in `EMSX_FIELD_DATA`.

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

Data Types: struct

### **timeout** — Timeout value

500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer to denote the time in milliseconds the default or custom event handler waits for an event in the current queue.

Data Types: double

### **options** — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure to reuse the settings for specifying a custom event handler or timeout value for the event handler.

Specify using a custom event handler and a timeout value of 500 milliseconds.

```
Example: options.useDefaultEventHandler = false;
options.timeOut = 500;
```

Data Types: struct

## Output Arguments

### **events** — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

### More About

#### Tips

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

- “Workflow for Bloomberg EMSX”
- “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17

#### See Also

`timer` | `createOrder` | `createOrderAndRouteWithStrat` | `delete` | `deleteOrder` | `getBrokerInfo` | `modifyRoute` | `orders` | `routeOrder` | `routes` | `start` | `stop`

# orders

Obtain Bloomberg EMSX order subscription

## Syntax

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

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

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

## Description

`[events,subs] = orders(c,fields)` subscribes to Bloomberg EMSX fields `fields` using the Bloomberg EMSX connection `c`. `orders` returns existing event data `events` from the event queue and the Bloomberg EMSX subscription list `subs`.

`[events,subs] = orders(c,fields,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments to specify a custom event handler or timeout value for the event handler.

`[events,subs] = orders(c,fields,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The `options` structure fields and values correspond to names and values of name-value pair arguments, respectively.

## Examples

### Subscribe to Order Events Using the Default Event Handler

Create the Bloomberg EMSX connection `c`.

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

Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c` and Bloomberg EMSX field list `fields`.

```
fields = {'EMSX_BROKER', 'EMSX_AMOUNT', 'EMSX_FILLED'};
[events,subs] = orders(c,fields)
events =
    MSG_TYPE: {'E'}
    MSG_SUB_TYPE: {'O'}
    EVENT_STATUS: 4
    ...
subs =
com.bloombergblp.blpapi.SubscriptionList@4bc3dc78
```

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

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

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Subscribe to Order Events Using the Custom Event Handler

Create the Bloomberg EMSX connection `c`.

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

Write a custom event handler function named `eventhandler`. Run the custom event handler using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17.

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

`t` is the timer object.

Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c` and Bloomberg EMSX field list `fields`. Use the custom event handler by setting the name-value pair argument `'useDefaultEventHandler'` to `false`.

```
fields = {'EMSX_BROKER', 'EMSX_AMOUNT', 'EMSX_FILLED'};
[events,subs] = orders(c,fields,'useDefaultEventHandler',false)

events =

    []

subs =
```

```
com.bloomberglp.blpapi.SubscriptionList@2c5b1c7e
```

`events` contains an empty double. The custom event handler processes the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from order events using the Bloomberg EMSX subscription list object `subs`. Stop the timer to stop data updates using `stop`.

```
b.Session.unsubscribe(subs)
stop(t)
```

Delete the timer if you are done processing data updates using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Subscribe to Order Events Using a Timeout

Create the Bloomberg EMSX connection `c`.

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

Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c` and Bloomberg EMSX field list `fields`. Specify the name-value pair argument `timeOut` and set it to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code.

```
fields = {'EMSX_BROKER', 'EMSX_AMOUNT', 'EMSX_FILLED'};
[events,subs] = orders(c,fields,'timeOut',200)

events =
```

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

```
subs =
```

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

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

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

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Subscribe to Order Events Using the Options Structure

Create the Bloomberg EMSX connection `c`.

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

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code. Subscribe to events for Bloomberg EMSX orders using the Bloomberg EMSX connection `c`, Bloomberg EMSX field list `fields`, and options structure `options`.

```
options.timeOut = 200;
options.useDefaultEventHandler = true;

fields = {'EMSX_BROKER', 'EMSX_AMOUNT', 'EMSX_FILLED'};

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

events =
```

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

```
subs =
```

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

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

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

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

### **fields** — Bloomberg EMSX field information

cell array

Bloomberg EMSX field information, specified using a cell array. Use `getAllFieldMetaData` to view available field information for the Bloomberg EMSX service.

```
Example: 'EMSX_TICKER'  
'EMSX_AMOUNT'  
'EMSX_ORDER_TYPE'
```

Data Types: cell

### **options** — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure instead of name-value pair arguments to reuse the optional name-value pair arguments to specify a custom event handler or timeout value for the event handler.

The `options` structure field and values correspond to names and values of the name-value pair arguments, respectively.

Specify using a custom event handler and a timeout value of 500 milliseconds.

```
Example: options.useDefaultEventHandler = false;  
options.timeOut = 500;
```

Data Types: `struct`

### Name-Value Pair Arguments

Specify optional 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: 'useDefaultEventHandler',false
```

#### 'useDefaultEventHandler' — Flag for event handler preference

true (default) | false

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, specified as the comma-separated pair consisting of 'useDefaultEventHandler' and the logical values true or false.

To specify the default event handler, set this flag to true.

Otherwise, set this flag to false to specify a custom event handler.

Data Types: `logical`

#### 'timeOut' — Timeout value for event handler

500 milliseconds (default) | nonnegative integer

Timeout value for event handler for the Bloomberg EMSX service, specified as the comma-separated pair consisting of 'timeOut' and a nonnegative integer in units of milliseconds.

```
Example: 'timeOut',200
```



Data Types: double

## Output Arguments

### **events** — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, **events** is a structure containing the current contents of the event queue. Otherwise, **events** is an empty double.

When the name-value pair argument 'useDefaultEventHandler' or the same field for the structure **options** is set to **false**, **events** is an empty double.

### **subs** — Bloomberg EMSX subscription list

subscription list object

Bloomberg EMSX subscription list, returned as a Bloomberg EMSX subscription list object.

## More About

### Tips

- For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.
- “Workflow for Bloomberg EMSX”
- “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17

### See Also

timer | close | createOrder | createOrderAndRoute | createOrderAndRouteWithStrat | delete | deleteOrder | deleteRoute | emsx | getAllFieldMetaData | modifyOrder | routeOrder | routes | start | stop

# emsxOrderBlotter

Bloomberg EMSX example order blotter

## Syntax

```
[t,subs] = emsxOrderBlotter(c)
```

## Description

`[t,subs] = emsxOrderBlotter(c)` displays a trader's order information. `c` is the Bloomberg EMSX connection, `t` is the timer object associated with the event handler, and `subs` is the Bloomberg EMSX subscription list.

## Examples

### Display the Order in an Order Blotter

Create the Bloomberg EMSX connection `c`.

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

Open Bloomberg EMSX order blotter using the Bloomberg EMSX connection `c`.

```
[t,subs] = emsxOrderBlotter(c)
```

```
Timer Object: timer-1
```

```
Timer Settings
```

```
  ExecutionMode: fixedRate
```

```
    Period: 1
```

```
    BusyMode: drop
```

```
    Running: on
```

```
Callbacks
```

```
  TimerFcn: {@processEventToBlotter [1x1 emsx]}
```

```
  ErrorFcn: ''
```

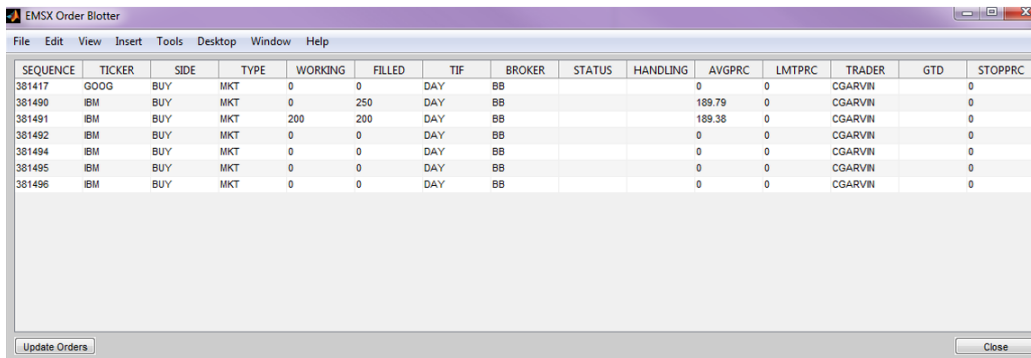
```
  StartFcn: ''
```

```
StopFcn: ''
```

```
subs =
```

```
com.bloomberg1p.blpapi.SubscriptionList@3e24da58
```

`emsxOrderBlotter` returns the timer object output and the Bloomberg EMSX subscription list object. For details about the timer object, see `timer`.



The screenshot shows a window titled "EMSX Order Blotter" with a menu bar (File, Edit, View, Insert, Tools, Desktop, Window, Help) and a table of order data. The table has the following columns: SEQUENCE, TICKER, SIDE, TYPE, WORKING, FILLED, TIF, BROKER, STATUS, HANDLING, AVGPCR, LMTPCR, TRADER, GTD, and STOPPCR. The data rows are as follows:

SEQUENCE	TICKER	SIDE	TYPE	WORKING	FILLED	TIF	BROKER	STATUS	HANDLING	AVGPCR	LMTPCR	TRADER	GTD	STOPPCR
381417	GOOG	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0
381490	IBM	BUY	MKT	0	250	DAY	BB			189.79	0	CGARVIN		0
381491	IBM	BUY	MKT	200	200	DAY	BB			189.38	0	CGARVIN		0
381492	IBM	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0
381494	IBM	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0
381495	IBM	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0
381496	IBM	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0

At the bottom of the window, there are two buttons: "Update Orders" and "Close".

The order blotter displays the current order information for a trader.

Create the order request structure `order` to define the order parameters. This code creates a buy market order for 330 shares of IBM. This code uses the broker `BB` with the time in force set to `DAY` and any hand instruction. Convert the number of shares to a 32-bit signed integer using `int32`.

```
order.EMSX_TICKER = 'IBM';
order.EMSX_AMOUNT = int32(330);
order.EMSX_ORDER_TYPE = 'MKT';
order.EMSX_BROKER = 'BB';
order.EMSX_TIF = 'DAY';
order.EMSX_HAND_INSTRUCTION = 'ANY';
order.EMSX_SIDE = 'BUY';
```

Create and route the order using the Bloomberg EMSX connection `c` and the order request structure `order`. Use the custom event handler `processEventToBlotter` by setting the name-value pair argument `'useDefaultEventHandler'` to `false`.

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

```

events =

    []

CreateOrderAndRoute = {

    EMSX_SEQUENCE = 381499

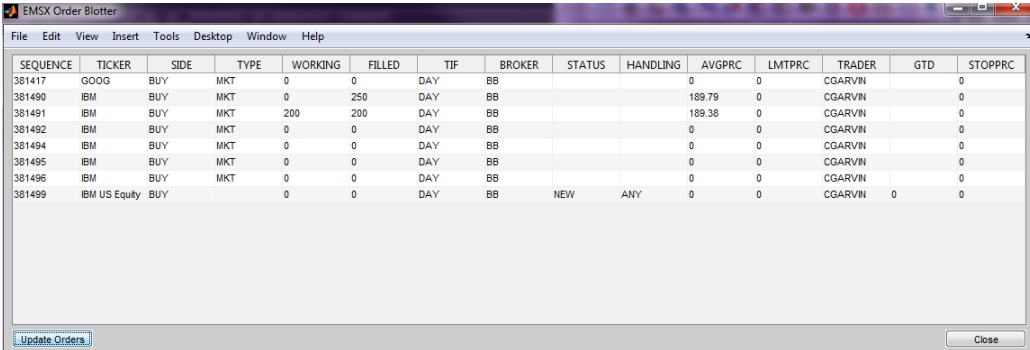
    EMSX_ROUTE_ID = 1

    MESSAGE = Order created and routed

}

```

`createOrderAndRoute` creates the order, routes the order, and returns a structure `events` that contains an empty double. `processEventToBlotter` displays output from `createOrderAndRoute` with the order number `EMSX_SEQUENCE`, route number `EMSX_ROUTE_ID`, and message: Order created and routed.



The screenshot shows the 'EMSX Order Blotter' window with a menu bar (File, Edit, View, Insert, Tools, Desktop, Window, Help) and a table of order data. The table has 16 columns: SEQUENCE, TICKER, SIDE, TYPE, WORKING, FILLED, TIF, BROKER, STATUS, HANDLING, AVGPCR, LMTPRC, TRADER, GTD, and STOPPRC. The data rows are as follows:

SEQUENCE	TICKER	SIDE	TYPE	WORKING	FILLED	TIF	BROKER	STATUS	HANDLING	AVGPCR	LMTPRC	TRADER	GTD	STOPPRC
381417	GOOG	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0
381490	IBM	BUY	MKT	0	250	DAY	BB			189.79	0	CGARVIN		0
381491	IBM	BUY	MKT	200	200	DAY	BB			189.38	0	CGARVIN		0
381492	IBM	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0
381494	IBM	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0
381495	IBM	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0
381496	IBM	BUY	MKT	0	0	DAY	BB			0	0	CGARVIN		0
381499	IBM US Equity	BUY		0	0	DAY	BB	NEW	ANY	0	0	CGARVIN	0	0

At the bottom of the window, there are two buttons: 'Update Orders' and 'Close'.

The order blotter updates using the information for the created and routed order, where order number `EMSX_SEQUENCE` is `381499`, using the event handler function `processEventToBlotter`. The order blotter updates as orders are created and managed.

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”

- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

## Output Arguments

### **t** — MATLAB timer

object

MATLAB timer, returned as a MATLAB object. For details, see `timer`.

### **subs** — Bloomberg EMSX subscription list

subscription list object

Bloomberg EMSX subscription list, returned as a Bloomberg EMSX subscription list object.

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”

### See Also

`timer` | `close` | `createOrder` | `createOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `deleteOrder` | `deleteRoute` | `emsx` | `modifyOrder` | `orders` | `routeOrder` | `routes`

## processEvent

Sample Bloomberg EMSX event handler

### Syntax

```
processEvent(c)
```

### Description

`processEvent(c)` displays and flushes the event queue associated with the Bloomberg EMSX connection `c`. `processEvent` is a sample event handler function. You can build a custom event handler function to process Bloomberg EMSX events.

### Examples

#### Continually Process the Bloomberg EMSX Event Queue

Create the Bloomberg EMSX connection `c`.

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

Use `timer` to continually process the Bloomberg EMSX event queue.

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

`t` is the MATLAB timer object. For details, see `timer`.

Close the Bloomberg EMSX connection.

```
close(c)
```

#### Process the Bloomberg EMSX Event Queue Once

Create the Bloomberg EMSX connection `c`.

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

Use the default event handler function `processEvent` to process the Bloomberg EMSX event queue once.

```
processEvent(c)
SessionConnectionUp = {
    server = "localhost/127.0.0.1:8194"
}
SessionStarted = {
}
ServiceOpened = {
    serviceName = "//blp/emapisvc_beta"
}
```

`processEvent` clears the Bloomberg EMSX event queue.

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

**c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”
- “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17

### **See Also**

timer | close | createOrder | createOrderAndRoute |  
createOrderAndRouteWithStrat | deleteOrder | deleteRoute | emsx |  
modifyOrder | orders | routeOrder | routes



# routeOrder

Route Bloomberg EMSX order

## Syntax

```
events = routeOrder(c,route)
events = routeOrder(c,route,'timeOut',timeout)

routeOrder( ____, 'useDefaultEventHandler', false)

____ = routeOrder(c,route,options)
```

## Description

`events = routeOrder(c,route)` routes a Bloomberg EMSX order using the Bloomberg EMSX connection `c` and route request `route`. `routeOrder` returns a status message using the default event handler.

`events = routeOrder(c,route,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`routeOrder( ____, 'useDefaultEventHandler', false)` routes a Bloomberg EMSX order using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`____ = routeOrder(c,route,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

## Examples

### Route an Order Using the Default Event Handler

To route a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Route the order using the Bloomberg EMSX connection `c` and `route`.

```
events = routeOrder(c,route)

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

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

### Route an Order Using a Timeout

To route a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Route the order using the Bloomberg EMSX connection `c` and `route`. Set the timeout value to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code.

```
events = routeOrder(c,route,'timeOut',200)
```

```
events =
```

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

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

### Route an Order Using a Custom Event Handler

To route a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17.

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

`t` is the MATLAB timer object. For details, see `timer`.

Route the order using the Bloomberg EMSX connection `c` and `route`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with routing an order.

```
routeOrder(c,route,'useDefaultEventHandler',false)
```

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

```
b.Session.unsubscribe(osubs)
b.Session.unsubscribe(rsubs)
stop(t)
```

Delete the timer if you are done processing data updates using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Route an Order Using an Options Structure

To route a Bloomberg EMSX order, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code. Route the order using the Bloomberg EMSX connection `c`, `route`, and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;
```

```
events = routeOrder(c,route,options)
```

```
events =
```

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

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

### **route** — Route request

structure

Route request, specified as a structure containing these fields.

Convert the numbers to 32-bit signed integers using `int32`. `EMSX_SEQUENCE` must denote an existing order sequence number.

Field	Description
<code>EMSX_SEQUENCE</code>	Bloomberg EMSX order sequence number

Field	Description
EMSX_TICKER	Bloomberg EMSX ticker symbol
EMSX_AMOUNT	Bloomberg EMSX number of shares
EMSX_BROKER	Bloomberg EMSX broker name
EMSX_HAND_INSTRUCTION	Bloomberg EMSX hand instruction

```
Example: route.EMSX_SEQUENCE = int32(728918);
route.EMSX_TICKER = 'XYZ';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Data Types: struct

#### **timeout** — Timeout value

500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer to denote the time in milliseconds the default or custom event handler waits for an event in the current queue.

Data Types: double

#### **options** — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure to reuse the settings for specifying a custom event handler or timeout value for the event handler.

Specify using a custom event handler and a timeout value of 500 milliseconds.

```
Example: options.useDefaultEventHandler = false;
options.timeOut = 500;
```

Data Types: struct

## Output Arguments

#### **events** — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”
- “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17

### See Also

`timer` | `close` | `createOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `delete` | `deleteOrder` | `deleteRoute` | `emsx` | `modifyOrder` | `orders` | `routeOrderWithStrat` | `routes` | `start` | `stop`



# routeOrderWithStrat

Route Bloomberg EMSX order with strategies

## Syntax

```
events = routeOrderWithStrat(c,route,strat)
events = routeOrderWithStrat(c,route,strat,'timeOut',timeout)

routeOrderWithStrat( ____, 'useDefaultEventHandler',false)

____ = routeOrderWithStrat(c,route,strat,options)
```

## Description

`events = routeOrderWithStrat(c,route,strat)` routes a Bloomberg EMSX order with strategies using the Bloomberg EMSX connection `c`, route request `route`, and strategy structure `strat`. `routeOrderWithStrat` returns the order sequence number, route number, and status message using the default event handler.

`events = routeOrderWithStrat(c,route,strat,'timeOut',timeout)` specifies a timeout value `timeout` for the execution of the default event handler.

`routeOrderWithStrat( ____, 'useDefaultEventHandler',false)` routes a Bloomberg EMSX order with strategies using any of the input arguments in the previous syntaxes and a custom event handler. Write a custom event handler to process the events associated with routing orders. This syntax does not have an output argument because the custom event handler processes the contents of the event queue. If you want to use the default event handler instead, set the flag `'useDefaultEventHandler'` to `true` and use the `events` output argument. By default, the flag `'useDefaultEventHandler'` is set to `true`.

`____ = routeOrderWithStrat(c,route,strat,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The available `options` structure fields are `timeOut` and `useDefaultEventHandler`. Use the `events` output argument when the flag `useDefaultEventHandler` is set to `true` and omit this output argument when `useDefaultEventHandler` is set to `false`.

## Examples

### Route an Order Using the Default Event Handler

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

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

Route the order using the Bloomberg EMSX connection `c`, `route`, and `strat`.

```
events = routeOrderWithStrat(c,route,strat)
```

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

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

### Route an Order Using a Timeout

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

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

Route the order using the Bloomberg EMSX connection `c`, `route`, and `strat`. Set the timeout value to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code.

```
events = routeOrderWithStrat(c,route,strat,'timeOut',200)
```

```
events =
```

```
    EMSX_SEQUENCE: 335877
```

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

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

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

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

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

Close the Bloomberg EMSX connection.

```
close(c)
```

### Route an Order Using a Custom Event Handler

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

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

Suppose you create a custom event handler function called `eventhandler` with input argument `c`. Run `eventhandler` using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17.

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

`t` is the MATLAB timer object. For details, see `timer`.

Route the order using the Bloomberg EMSX connection `c`, `route`, and `strat`. Set the flag `'useDefaultEventHandler'` to `false` so that `eventhandler` processes the events associated with routing an order.

```
routeOrderWithStrat(c,route,strat,'useDefaultEventHandler',false)
```

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

```
b.Session.unsubscribe(osubs)
b.Session.unsubscribe(rsubs)
stop(t)
```

Delete the timer if you are done processing data updates using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Route an Order Using an Options Structure

To route a Bloomberg EMSX order with strategies, create the connection `c` using `emsx`, set up the order subscription using `orders`, and create the order using `createOrder`. For an example showing these activities, see “Create and Manage a Bloomberg EMSX Order”. Set up the route subscription using `routes`.

Define the route request structure `route`. Convert the numbers to 32-bit signed integers using `int32`. This code specifies to route 100 shares of IBM to the broker `BMTB` using any hand instruction and the order number `335877`.

```
route.EMSX_SEQUENCE = int32(335877);
```

```
route.EMSX_TICKER = 'IBM';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BMTB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Create the order strategies structure `strat` using the strategy `SSP`. Convert the field indicators to a 32-bit signed integer using `int32`.

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

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code. Route the order using the Bloomberg EMSX connection `c`, `route`, `strat`, and `options` structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;
```

```
events = routeOrderWithStrat(c, route, strat, options)
```

```
events =
```

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

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

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

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

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

Close the Bloomberg EMSX connection.

`close(c)`

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

### **route** — Route request

structure

Route request, specified as a structure containing these fields.

Convert the numbers to 32-bit signed integers using `int32`. `EMSX_SEQUENCE` must denote an existing order sequence number.

Field	Description
<code>EMSX_SEQUENCE</code>	Bloomberg EMSX order sequence number
<code>EMSX_TICKER</code>	Bloomberg EMSX ticker symbol
<code>EMSX_AMOUNT</code>	Bloomberg EMSX number of shares
<code>EMSX_BROKER</code>	Bloomberg EMSX broker name
<code>EMSX_HAND_INSTRUCTION</code>	Bloomberg EMSX hand instruction

```
Example: route.EMSX_SEQUENCE = int32(728918);
route.EMSX_TICKER = 'XYZ';
route.EMSX_AMOUNT = int32(100);
route.EMSX_BROKER = 'BB';
route.EMSX_HAND_INSTRUCTION = 'ANY';
```

Data Types: struct

### **strat** — Order strategies

structure

Order strategies, specified as a structure that contains the fields: `EMSX_STRATEGY_NAME`, `EMSX_STRATEGY_FIELD_INDICATORS`, and `EMSX_STRATEGY_FIELDS`. The structure field values must align with the strategy fields specified by `EMSX_STRATEGY_NAME`. For details about strategy fields and ordering, see `getBrokerInfo`.

Convert `EMSX_STRATEGY_FIELD_INDICATORS` to a 32-bit signed integer using `int32`. Set `EMSX_STRATEGY_FIELD_INDICATORS` equal to 0 for each field to use the field data setting in `EMSX_FIELD_DATA`. Or, set `EMSX_STRATEGY_FIELD_INDICATORS` equal to 1 to ignore the data in `EMSX_FIELD_DATA`.

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

Data Types: struct

### **timeout** — Timeout value

500 milliseconds (default) | nonnegative integer

Timeout value, specified as a nonnegative integer to denote the time in milliseconds the default or custom event handler waits for an event in the current queue.

Data Types: double

### **options** — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure to reuse the settings for specifying a custom event handler or timeout value for the event handler.

Specify using a custom event handler and a timeout value of 500 milliseconds.

```
Example: options.useDefaultEventHandler = false;
options.timeOut = 500;
```

Data Types: struct

## Output Arguments

### **events** — Event queue contents

double | structure



Event queue contents, returned as a double or structure.

If the event queue contains events, `events` is a structure containing the current contents of the event queue. Otherwise, `events` is an empty double.

## More About

### Tips

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

- “Workflow for Bloomberg EMSX”
- “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17

### See Also

`timer` | `close` | `createOrder` | `createOrderAndRoute` | `createOrderAndRouteWithStrat` | `delete` | `deleteOrder` | `deleteRoute` | `emsx` | `getBrokerInfo` | `modifyOrder` | `orders` | `routeOrder` | `routes` | `start` | `stop`

# routes

Obtain Bloomberg EMSX route subscription

## Syntax

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

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

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

## Description

`[events,subs] = routes(c,fields)` subscribes to Bloomberg EMSX fields `fields` using the Bloomberg EMSX connection `c`. `routes` returns existing event data `events` from the event queue and the Bloomberg EMSX subscription list `subs`.

`[events,subs] = routes(c,fields,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments to specify a custom event handler or timeout value for the event handler.

`[events,subs] = routes(c,fields,options)` uses the `options` structure to customize the output, which is useful to preconfigure and save your options for repeated use. The `options` structure fields and values correspond to names and values of name-value pair arguments, respectively.

## Examples

### Set Up Route Subscription Using the Default Event Handler

Create the Bloomberg EMSX connection `c`.

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

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

```

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

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

events =

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

subs =

com.bloomberglp.blpapi.SubscriptionList@463b9287

```

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

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

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Set Up Route Subscription Using a Custom Event Handler

Create the Bloomberg EMSX connection `c`.

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

Write a custom event handler function named `eventhandler`. Run the custom event handler using `timer`. Start the timer to run `eventhandler` immediately using `start`. For details, see “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17.

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

`t` is the timer object.

Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`. Use the custom event handler by setting the name-value pair argument `'useDefaultEventHandler'` to `false`.

```
fields = {'EMSX_BROKER', 'EMSX_WORKING'};
[events,subs] = routes(c,fields,'useDefaultEventHandler',false)
events =
    []
subs =
com.bloomberglp.blpapi.SubscriptionList@463b9287
```

`events` is an empty double. The custom event handler processes the event queue. `subs` contains the Bloomberg EMSX subscription list object.

Unsubscribe from route events using the Bloomberg EMSX subscription list object `subs`. Stop the timer to stop data updates using `stop`.

```
b.Session.unsubscribe(subs)
stop(t)
```

Delete the timer if you are done processing data updates using `delete`.

```
delete(t)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### **Set Up Route Subscription Using a Timeout**

Create the Bloomberg EMSX connection `c`.

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

Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c`. Specify the name-value pair argument `timeOut` and set it to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code.

```
fields = {'EMSX_BROKER', 'EMSX_WORKING'};
[events,subs] = routes(c,fields,'timeOut',200)
events =
```

```

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

```

```
subs =
```

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

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

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

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

### Set Up Route Subscription Using an Options Structure

Create the Bloomberg EMSX connection `c`.

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

Create a structure `options`. To use the default event handler, set the field `useDefaultEventHandler` to `true`. Set the field `timeOut` to 200 milliseconds. `timeOut` specifies how long the event handler listens to the queue for an event for each iteration of the code. Set up the route subscription for Bloomberg EMSX fields `EMSX_BROKER` and `EMSX_WORKING` using the Bloomberg EMSX connection `c` and options structure `options`.

```
options.useDefaultEventHandler = true;
options.timeOut = 200;
fields = {'EMSX_BROKER', 'EMSX_WORKING'};
```

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

```
events =
```

```

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

```

```
subs =
```

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

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

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

```
b.Session.unsubscribe(subs)
```

Close the Bloomberg EMSX connection.

```
close(c)
```

- “Create and Manage a Bloomberg EMSX Order”
- “Create and Manage a Bloomberg EMSX Route”
- “Manage a Bloomberg EMSX Order and Route”

## Input Arguments

### **c** — Bloomberg EMSX service connection

connection object

Bloomberg EMSX service connection, specified as a connection object created using `emsx`.

### **fields** — Bloomberg EMSX field information

cell array

Bloomberg EMSX field information, specified using a cell array. Use `getAllFieldMetaData` to view available field information for the Bloomberg EMSX service.

```
Example: 'EMSX_TICKER'  
'EMSX_AMOUNT'  
'EMSX_ORDER_TYPE'
```

Data Types: cell

### **options** — Options for custom event handler or timeout value

structure

Options for custom event handler or timeout value, specified as a structure. Use the options structure instead of name-value pair arguments to reuse the optional name-value pair arguments to specify a custom event handler or timeout value for the event handler.

The `options` structure field and values correspond to names and values of the name-value pair arguments, respectively.

Specify using a custom event handler and a timeout value of 500 milliseconds.

```
Example: options.useDefaultEventHandler = false;  
options.timeOut = 500;
```

Data Types: `struct`

## Name-Value Pair Arguments

Specify optional 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: 'useDefaultEventHandler',false
```

### 'useDefaultEventHandler' — Flag for event handler preference

true (default) | false

Flag for event handler preference, indicating whether to use the default or custom event handler to process order events, specified as the comma-separated pair consisting of 'useDefaultEventHandler' and the logical values true or false.

To specify the default event handler, set this flag to true.

Otherwise, set this flag to false to specify a custom event handler.

Data Types: `logical`

### 'timeOut' — Timeout value for event handler

500 milliseconds (default) | nonnegative integer

Timeout value for event handler for the Bloomberg EMSX service, specified as the comma-separated pair consisting of 'timeOut' and a nonnegative integer in units of milliseconds.

```
Example: 'timeOut',200
```

Data Types: double

## Output Arguments

### **events** — Event queue contents

double | structure

Event queue contents, returned as a double or structure.

If the event queue contains events, **events** is a structure containing the current contents of the event queue. Otherwise, **events** is an empty double.

When the name-value pair argument 'useDefaultEventHandler' or the same field for the structure **options** is set to **false**, **events** is an empty double.

### **subs** — Bloomberg EMSX subscription list

subscription list object

Bloomberg EMSX subscription list, returned as a Bloomberg EMSX subscription list object.

## More About

### Tips

- For details about Bloomberg EMSX, see the *EMSX API Programmer's Guide* using the **WAPI <GO>** option from the Bloomberg terminal.
- Suppose you create a custom event handler function called **eventhandler** with input argument **c**. Run **eventhandler** using this code.

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

**t** is the MATLAB timer object. For details, see **timer**.

- “Workflow for Bloomberg EMSX”
- “Writing and Running Custom Event Handler Functions with Bloomberg EMSX” on page 1-17



**See Also**

timer | close | createOrder | createOrderAndRoute |  
createOrderAndRouteWithStrat | delete | deleteOrder | deleteRoute | emsx  
| getAllFieldMetaData | modifyOrder | modifyRoute | orders | routeOrder |  
start | stop

# xtrdr

Create X\_TRADER connection

## Syntax

```
X = xtrdr
```

## Description

X = xtrdr starts X\_TRADER or connects to an existing X\_TRADER session.

## Examples

### Create a Connection to X\_TRADER

```
X = xtrdr
```

```
x =
```

```
    xtrdr with properties:
```

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

- “X\_TRADER Price Update”
- “X\_TRADER Price Update Depth”
- “X\_TRADER Order Submission”

## Output Arguments

**X** — X\_TRADER connection

connection object

X\_TRADER connection, returned as a connection object for an 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.

## More About

- “Workflows for Trading Technologies X\_TRADER”
- X\_TRADER API

## See Also

close

# close

Close X\_TRADER connection

## Syntax

```
close(X)
```

## Description

`close(X)` closes the X\_TRADER connection X.

## Examples

### Close X\_TRADER Connection

```
close(X)
```

- “X\_TRADER Price Update”
- “X\_TRADER Price Update Depth”
- “X\_TRADER Order Submission”

## Input Arguments

### X — X\_TRADER connection

connection object

X\_TRADER connection, specified as a connection object created using `xtrdr`.

## More About

- “Workflows for Trading Technologies X\_TRADER”
- X\_TRADER API

**See Also**  
xtrdr

# createInstrument

Create instrument for X\_TRADER

## Syntax

```
createInstrument(c,s)  
createInstrument(c,Name,Value)
```

## Description

`createInstrument(c,s)` creates the X\_TRADER instrument defined by the structure `s` with fields corresponding to valid X\_TRADER API options. For details, see the [Trading Technologies X\\_TRADER API Programming Tutorial](#) or [X\\_TRADER API Class Reference](#).

`createInstrument(c,Name,Value)` creates the instrument using one or more `Name,Value` pair arguments with names and values corresponding to valid X\_TRADER API options. For details, see the [Trading Technologies X\\_TRADER API Programming Tutorial](#) or [X\\_TRADER API Class Reference](#).

## Examples

### Create an X\_TRADER Instrument Using an Input Structure

The instruments used in these examples continually expire. To ensure you use a current instrument, see the **Market Explorer** in X\_TRADER Pro.

Create the X\_TRADER connection.

```
c = xtrdr;
```

Define an input structure `s` with fields corresponding to valid X\_TRADER API options. For example, create the input structure for Euro-Bobl Futures.

```
s = [];  
s.Exchange = 'Eurex';  
s.Product = 'OGBM';
```

```

s.ProdType = 'Option';
s.Contract = 'Jan12 P12300';
s.Alias = 'TestInstrument3';
s
s =

    Exchange: 'Eurex'
    Product: 'OGBM'
    ProdType: 'Option'
    Contract: 'Jan12 P12300'
    Alias: 'TestInstrument3'

```

---

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

---

Create an X\_TRADER instrument.

```
createInstrument(c,s)
```

Close the connection.

```
close(c)
```

### Create an X\_TRADER Instrument Using Name-Value Pairs

Create the X\_TRADER connection.

```
c = xtrdr;
```

Create an X\_TRADER instrument for Euro-Bobl Futures using name-value pair arguments corresponding to valid X\_TRADER API options.

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

Close the connection.

```
close(c)
```

### Retrieve Data Using Multiple X\_TRADER Instruments

Create the X\_TRADER connection.

```
c = xtrdr;
```

Create an X\_TRADER instrument for Euro-Bobl Futures using name-value pair arguments corresponding to valid X\_TRADER API options.

```
createInstrument(c, 'Exchange', 'Eurex', 'Product', 'OGBM', ...  
                'ProdType', 'Option', 'Contract', 'Jun14 P127', ...  
                'Alias', 'PriceInstrumentEurex')
```

Create another X\_TRADER instrument for CAISO NP15 EZ Gen Hub 5 MW Peak Calendar-Day Real-Time LMP Futures using name-value pair arguments corresponding to valid X\_TRADER API options. This contract expires in April 2014.

```
createInstrument(c, 'Exchange', 'CME', 'Product', '2F', ...  
                'ProdType', 'Future', 'Contract', 'Apr14', ...  
                'Alias', 'PriceInstrumentCMEApr14')
```

Create another X\_TRADER instrument for CAISO NP15 EZ Gen Hub 5 MW Peak Calendar-Day Real-Time LMP Futures using name-value pair arguments corresponding to valid X\_TRADER API options. This contract expires in October 2014.

```
createInstrument(c, 'Exchange', 'CME', 'Product', '2F', ...  
                'ProdType', 'Future', 'Contract', 'Oct14', ...  
                'Alias', 'PriceInstrumentCMEOct14')
```

Retrieve the exchange and product identifier for all three X\_TRADER instruments.

```
d = getData(c, {'Exchange', 'Product'})
```

```
d =  
    Exchange: {3x1 cell}  
    Product:  {3x1 cell}
```

d is a structure containing the Exchange and Product fields. The fields are cell arrays.

Display the Exchange field.

```
d.Exchange
```

```
ans =  
    'Eurex'  
    'CME'  
    'CME'
```

The Exchange field contains the exchange names Eurex and CME for the three X\_TRADER instruments.

Close the connection.



`close(c)`

- “X\_TRADER Price Update”
- “X\_TRADER Price Update Depth”
- “X\_TRADER Order Submission”

## Input Arguments

### **c** – X\_TRADER connection

connection object

X\_TRADER connection, specified as a connection object created using `xtrdr`.

### **s** – X\_TRADER input structure

structure

X\_TRADER input structure, specified using fields corresponding to valid X\_TRADER API options. For details, see the Trading Technologies X\_TRADER API Programming Tutorial or X\_TRADER API Class Reference.

---

**Caution:** If the symbols for the exchange are entered incorrectly or the exchange server is down, an error appears. For example, if the exchange is “CME” and the CME exchange server is down, then this error appears: The price server for the Exchange CME is down. Unable to create instrument.

---

```
Example: s = [];  
s.Exchange = 'Eurex';  
s.Product = 'OGBM';  
s.ProdType = 'Option';  
s.Contract = 'Jan12 P12300';  
s.Alias = 'TestInstrument3';
```

Data Types: struct

## Name-Value Pair Arguments

Specify optional 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', 'Option', 'Co  
P12300', 'Alias', 'TestInstrument3')
```

### 'Property1' — Valid X\_TRADER API options

string

Valid X\_TRADER API options, specified using information in the Trading Technologies X\_TRADER API Programming Tutorial or X\_TRADER API Class Reference.

---

#### Requirements:

- When using the 'Alias' name-value pair argument, ensure that every 'Alias' name is unique across all X\_TRADER instruments.
- Restart the MATLAB session before reusing an 'Alias' name.

Otherwise, `createInstrument` returns an error.

---

Data Types: char

### 'Property2' — Valid X\_TRADER API options

string

Valid X\_TRADER API options, specified using information in the Trading Technologies X\_TRADER API Programming Tutorial or X\_TRADER API Class Reference.

Data Types: char

## More About

- “Workflows for Trading Technologies X\_TRADER”
- X\_TRADER API

## See Also

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

# createNotifier

Create instrument notifier for X\_TRADER

## Syntax

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

## Description

`createNotifier(X,S)` creates the `xtrdr` instrument notifier defined by the structure `S` with fields corresponding to valid `X_TRADER` API options. For details, see the *Trading Technologies X\_TRADER API 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*.

## 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 = '1';  
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)
```

- “X\_TRADER Price Update”

- “X\_TRADER Price Update Depth”
- “X\_TRADER Order Submission”

## Input Arguments

### X — X\_TRADER connection

connection object

X\_TRADER connection, specified as a connection object created using `xtrdr`.

### S — xtrdr input structure with fields

structure

`xtrdr` input structure, specified with fields corresponding to valid X\_TRADER API options. For details, see the Trading Technologies *X\_TRADER API 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',-1,'EnableDepthUpdates',0,'DebugLogL
[],'DeliverAllPriceUpdates',0)
```

### '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', -1, 'EnableDepthUpdates', 0, 'DebugLogL  
[], 'DeliverAllPriceUpdates', 0)
```

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', -1, 'EnableDepthUpdates', 0, 'DebugLogL  
[], 'DeliverAllPriceUpdates', 0)
```

Data Types: char

## More About

- “Workflows for Trading Technologies X\_TRADER”
- X\_TRADER API

## See Also

[createInstrument](#) | [createOrderProfile](#) | [createOrderSet](#) | [xtrdr](#)

# createOrderProfile

Create order profile for X\_TRADER

## Syntax

```
P = createOrderProfile(X,S)
P = createOrderProfile(X,Name,Value)
```

## Description

`P = createOrderProfile(X,S)` creates an order profile defined by the structure `S` with fields corresponding to valid `X_TRADER` API options. For details, see the Trading Technologies *X\_TRADER API 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*.

## 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)
```

- “X\_TRADER Price Update”
- “X\_TRADER Price Update Depth”
- “X\_TRADER Order Submission”



## Input Arguments

### **X** — **X\_TRADER** connection

connection object

X\_TRADER connection, specified as a connection object created using `xtrdr`.

### **S** — **xtrdr** input structure with fields

structure

`xtrdr` input structure, specified with fields corresponding to valid X\_TRADER API options. For details, see the Trading Technologies *X\_TRADER API 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', '', 'RoundOption', 2, 'CustomerDefaults')
```

### **'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', '', 'RoundOption', 2, 'CustomerDefaults')
```

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', '', 'RoundOption', 2, 'CustomerDefaults')
```

Data Types: char

## Output Arguments

### P — Order profile

structure

Order profile, returned as a structure.

## More About

- “Workflows for Trading Technologies X\_TRADER”
- X\_TRADER API

## See Also

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

# createOrderSet

Create order set for X\_TRADER

## Syntax

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

## Description

`createOrderSet(X)` creates an `xtrdr` order set with empty properties. You can set the properties individually using X\_TRADER API options. For details, see the *Trading Technologies X\_TRADER API 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*.

## 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)
```

- “X\_TRADER Price Update”
- “X\_TRADER Price Update Depth”
- “X\_TRADER Order Submission”

## 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', 'WriteProperties',  
[] 'DebugLogLevel', 3, 'QuoteWithCancelReplace', 0, 'EnableOrderUpdateData', 0, 'Enabl
```

#### '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

### More About

- “Workflows for Trading Technologies X\_TRADER”
- X\_TRADER API

### See Also

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

## getData

Obtain current X\_TRADER data

### Syntax

```
D = getData(X,S,F)
D = getData(X,F)
```

### Description

`D = getData(X,S,F)` returns data for the fields `F` for the `xtrdr` instrument object, `S`, with fields corresponding to valid X\_TRADER API options. For details, see the *Trading Technologies X\_TRADER API 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`.

### 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}
```

- “X\_TRADER Price Update”
- “X\_TRADER Price Update Depth”
- “X\_TRADER Order Submission”

## Input Arguments

### X — X\_TRADER connection

connection object

X\_TRADER connection, specified as a connection object created 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.

## More About

- “Workflows for Trading Technologies X\_TRADER”
- X\_TRADER API

## See Also

`createInstrument` | `xtrdr`

### **cqg**

Create CQG connection object

### **Syntax**

```
c = cqg
```

### **Description**

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

### **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.ICQGAPIConfig]
```

CQG connection object properties reflect the CQG ActiveX object `Handle` and the API configuration type library specification `APIConfig`.

Display the `Handle` property of `c`.

```
c.Handle
```

```
ans =
```

```
    COM.CQG_CQGCEL_4
```

Close the CQG connection.

`close(c)`

- “Create CQG Order”
- “Request CQG Historical Data”
- “Request CQG Intraday Tick Data”
- “Request CQG Real-Time Data”

## 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.

## More About

- “Workflow for CQG”
- CQG API Reference Guide

## See Also

`close` | `startUp`

# close

Close CQG connection

## Syntax

```
close(c)
```

## Description

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

## Examples

### Close the CQG Connection

Create the CQG connection object `c` using `cqg`.

```
c = cqg;
```

Create the CQG connection using `startUp`.

```
startUp(c)
```

Close the connection using the CQG connection object `c`.

```
close(c)
```

- “Create CQG Order”
- “Request CQG Historical Data”
- “Request CQG Intraday Tick Data”
- “Request CQG Real-Time Data”

## Input Arguments

**c** — **CQG connection**  
connection object

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

## More About

- “Workflow for CQG”
- CQG API Reference Guide

## See Also

`cqg` | `shutDown`

# createOrder

Create CQG order

## Syntax

```
o = createOrder(c,s,1,account,quantity)
o = createOrder(c,s,2,account,quantity,limitprice)
o = createOrder(c,s,3,account,quantity,stopprice)
o = createOrder(c,s,4,account,quantity,limitprice,stopprice)
```

## Description

`o = createOrder(c,s,1,account,quantity)` creates a `CQGOrder` object `o` for a market order of `quantity` shares of `CQG` instrument `s` using the `CQGAcount` credentials object `account` over the `CQG` connection `c`.

`o = createOrder(c,s,2,account,quantity,limitprice)` creates a limit order using a `CQG` limit price `limitprice`.

`o = createOrder(c,s,3,account,quantity,stopprice)` creates a stop order using a `CQG` stop price `stopprice`.

`o = createOrder(c,s,4,account,quantity,limitprice,stopprice)` creates a stop limit order using `CQG` limit and stop prices, `limitprice` and `stopprice`.

## Examples

### Create and Place a Market Order Using a CQGInstrument Object

To create and place a market order for shares of an instrument with the `CQG` Trader Com API using a `CQGInstrument` object to specify the instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with the connection status. Set up the API configuration properties. Then, register event handlers for tracking events associated with the instrument subscription, order and account. Subscribe to the instrument and create the `CQGInstrument` object `cqgInst`.

Then, set up the account credentials `accountHandle`. For an example demonstrating these activities, see “Create CQG Order”. 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”. 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”. 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”. 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”. 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)
```

- “Create CQG Order”

## 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*.

### More About

- “Workflow for CQG”
- CQG API Reference Guide

### See Also

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

# history

Request CQG historical data

## Syntax

```
history(c,s,startdate,enddate,period)
history(c,s,startdate,enddate,period,x)
```

## Description

`history(c,s,startdate,enddate,period)` requests CQG historical data asynchronously with bar size `period` between `startdate` and `enddate` for CQG instrument name `s` with CQG connection `c`.

`history(c,s,startdate,enddate,period,x)` requests CQG historical data asynchronously with additional request properties `x`.

## Examples

### Request CQG Historical Data

To request daily historical data for an instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see “Request CQG Historical Data”. 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”. 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)
```

- “Request CQG Historical Data”

## 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

## More About

- “Workflow for CQG”



- [CQG API Reference Guide](#)

**See Also**

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

## realtime

Subscribe to CQG instrument

### Syntax

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

### Description

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

### Examples

#### Subscribe to the CQG Instrument

To subscribe to the CQG instrument and get current data, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with instrument subscription. For an example demonstrating these activities, see “Request CQG Real-Time Data”. 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]
    [-2.1475e+09]
    [    660.5000]
    [-2.1475e+09]

```

Close the CQG connection.

```
close(c)
```

- “Request CQG Real-Time Data”

## 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

### More About

- “Workflow for CQG”
- CQG API Reference Guide

### See Also

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

# shutDown

Close CQG connection

## Syntax

```
shutDown(c)
```

## Description

shutDown(c) closes the CQG connection c.

## Examples

### Close the CQG Connection

Create the CQG connection object using `cqg`.

```
c = cqg;
```

Create the CQG connection using `startUp`.

```
startUp(c)
```

Close the CQG connection.

```
shutDown(c)
```

Alternatively, close the CQG connection using `close`.

```
close(c)
```

- “Create CQG Order”
- “Request CQG Historical Data”
- “Request CQG Intraday Tick Data”
- “Request CQG Real-Time Data”

### Input Arguments

**c** — **CQG connection**

connection object

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

### More About

- “Workflow for CQG”
- CQG API Reference Guide

### See Also

`close` | `cqg` | `startUp`

# startUp

Create CQG connection

## Syntax

```
startUp(c)
```

## Description

startUp(c) creates the CQG connection c.

## Examples

### Create the CQG Connection

Create the CQG connection object using `cqg`.

```
c = cqg;
```

Create the CQG connection.

```
startUp(c)
```

Close the CQG connection.

```
close(c)
```

- “Create CQG Order”
- “Request CQG Historical Data”
- “Request CQG Intraday Tick Data”
- “Request CQG Real-Time Data”

## Input Arguments

**c** — CQG connection  
connection object

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

### More About

- “Workflow for CQG”
- CQG API Reference Guide

### See Also

`close` | `cqg` | `shutDown`



## timeseries

Request CQG intraday tick data

### Syntax

```
timeseries(c,s,startdate,enddate)
timeseries(c,s,startdate,enddate,[ ],x)

timeseries(c,s,startdate,enddate,intraday)
timeseries(c,s,startdate,enddate,intraday,x)
```

### Description

`timeseries(c,s,startdate,enddate)` requests CQG raw intraday tick data asynchronously between `startdate` and `enddate` for CQG instrument name `s` with CQG connection `c`.

`timeseries(c,s,startdate,enddate,[ ],x)` requests CQG raw intraday tick data asynchronously without timed bar data using additional request properties `x`.

`timeseries(c,s,startdate,enddate,intraday)` requests CQG timed bar data asynchronously with the aggregated bar value `intraday`.

`timeseries(c,s,startdate,enddate,intraday,x)` requests CQG timed bar data asynchronously with additional request properties `x`.

### Examples

#### Request CQG Intraday Tick Data

To request intraday tick data for an instrument, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see “Request CQG Intraday Tick Data”. 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”. 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”. 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  -2.1475  -2.1475
    0.0007  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475
    0.0007  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475
    0.0007  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475
    0.0007  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475  -2.1475
    ...
```

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

Close the CQG connection.

```
close(c)
```

### Request CQG Timed Bar Data with Additional Properties

To request timed bar data for an instrument with an additional property, create the connection `c` using `cqg` and `startUp`. Register an event handler for tracking events associated with connection status. Set up the API configuration properties. Then, register an event handler for tracking events associated with building and initializing the output data structure. For an example demonstrating these activities, see “Request CQG Intraday Tick Data”. 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 -2.1475 -2.1475
  0.0007 -2.1475 -2.1475 -2.1475 -2.1475 -2.1475 -2.1475 -2.1475 -2.1475
  0.0007 -2.1475 -2.1475 -2.1475 -2.1475 -2.1475 -2.1475 -2.1475 -2.1475
  0.0007 -2.1475 -2.1475 -2.1475 -2.1475 -2.1475 -2.1475 -2.1475 -2.1475
  ...
```

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

Close the CQG connection.

`close(c)`

- “Request CQG Intraday Tick Data”

## 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

**More About**

- “Workflow for CQG”
- CQG API Reference Guide

**See Also**

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

# ibtws

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`.

## 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 port number that you choose.

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 for this example 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 IP address that you choose, and the port number that you choose.

Display the `Handle` property of `ib`.

```
ib.Handle
```

```
ans =
```

```
COM.TWS_TwsCtrl_1
```

Close the IB Trader Workstation connection.

```
close(ib)
```

- “Create Interactive Brokers Order”

- “Request Interactive Brokers Historical Data”
- “Request Interactive Brokers Real-Time Data”

## 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

The Interactive Brokers API determines these properties.

## More About

### Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:
  - Connection
  - Information resulting from executing functions
  - Errors
- “Workflow for Interactive Brokers”
- Interactive Brokers API Reference Guide

### See Also

`close`

# close

Close IB Trader Workstation connection

## Syntax

```
close(ib)
```

## Description

`close(ib)` closes the IB Trader Workstation connection `ib`.

## 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)
```

- “Create Interactive Brokers Order”
- “Request Interactive Brokers Historical Data”
- “Request Interactive Brokers Real-Time 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`.

## More About

- “Workflow for Interactive Brokers”
- Interactive Brokers API Reference Guide

## See Also

`ibtws`

# createOrder

Create IB Trader Workstation order

## Syntax

```
d = createOrder(ib,ibContract,ibOrder,id)
d = createOrder(ib,ibContract,ibOrder,id,eventhandler)
```

## Description

`d = createOrder(ib,ibContract,ibOrder,id)` creates an IB Trader Workstation order over the IB Trader Workstation connection `ib` using the IB Trader Workstation `IOrder` object `ibOrder` with a unique order identifier `id` to denote the order information. `createOrder` uses the IB Trader Workstation `IContract` object `ibContract` to signify the instrument for the transaction. `createOrder` returns the Interactive Brokers order data `d` containing data about the completed order.

`d = createOrder(ib,ibContract,ibOrder,id,eventhandler)` creates an IB Trader Workstation order using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

## Examples

### Create an Order

To create an order, set up the IB Trader Workstation connection `ib` using `ibtw`. Create an IB Trader Workstation `IContract` object `ibContract`. An `IContract` object is an Interactive Brokers object for containing the 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 order conditions to place an order. For an example showing how to create these objects, see “Create Interactive Brokers Order”. For details about creating these objects, see Interactive Brokers API Reference Guide.

Obtain the next valid order identification number `id` using `ib`.

```
id = orderid(ib)
```

```
id =
```

```
54110686
```

Execute the order using `ib`, `ibContract`, `ibOrder`, and `id`. This code assumes a buy market order for two shares.

```
d = createOrder(ib,ibContract,ibOrder,id)
```

```
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 these fields:

- Status
- Filled
- Remaining
- Average fill price
- Permanent identifier
- Parent identifier
- Last fill price
- Client identifier
- 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)
```

### Create an Order Using an Event Handler

To create an order, set up the IB Trader Workstation connection `ib` using `ibtw`. Create an IB Trader Workstation `IContract` object `ibContract`. An `IContract` object is an Interactive Brokers object for containing the 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 order conditions to place an order. For an example showing how to create these objects, see “Create Interactive Brokers Order”. For details about creating these objects, see Interactive Brokers API Reference Guide.

Obtain the next valid order identification number `id` using `ib`.

```
id = orderid(ib)
```

```
id =
```

```
768409.00
```

Execute the order using `ib`, `ibContract`, `ibOrder`, and `id`. This code assumes a buy market order for two shares. Use the sample event handler function `ibExampleEventHandler` or write a custom event handler function.

```
d = createOrder(ib,ibContract,ibOrder,id,@ibExampleEventHandler)
```

```
d =
```

```
768409.00
```

```
Columns 1 through 5
```

```
[1x1 COM.TWS_TwsCtrl_1] [13.00] [768409.00] 'Submitted' [0]
```

```
Columns 6 through 12
```

```
[2.00] [0] [1679681704.00] [0] [0] [0] ''
```

```
Columns 13 through 14
```

```
[1x1 struct] 'orderStatus'
```



...

`d` contains the unique order identifier `id`.

`ibExampleEventHandler` displays order status data in the Command Window. The columns are:

- Interactive Brokers ActiveX object
- Event identifier
- Unique order identifier
- Order status
- Filled
- Remaining
- Average fill price
- Permanent identifier
- Parent identifier
- Last fill price
- Client identifier
- Why held
- Structure that repeats the contents of the columns
- Event type

For details about this data, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(ib)
```

- “Create Interactive Brokers 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. Create this object by calling the Interactive Brokers API function `createContract`. For details about `createContract` and the attributes that 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. Create this object by calling the Interactive Brokers API function `createOrder`. For details about the attributes that you can set and `createOrder`, see Interactive Brokers API Reference Guide.

### **id** — IB Trader Workstation order unique identifier

scalar

IB Trader Workstation order unique identifier, specified as a scalar.

Data Types: `double`

### **eventhandler** — Event handler

function handle | string

Event handler, specified as a function handle or a string to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

## Output Arguments

### **d** — Interactive Brokers order data

structure | double

Interactive Brokers order data, returned as a structure containing these fields:

- Status
- Filled
- Remaining
- Average fill price
- Permanent identifier
- Parent identifier
- Last fill price
- Client identifier
- Why held

When using an event handler function, `d` is a double containing the unique order identifier.

## More About

### Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:
  - Connection
  - Information resulting from executing functions
  - Errors
- “Workflow for Interactive Brokers”
- “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20
- Interactive Brokers API Reference Guide

### See Also

`close` | `getdata` | `history` | `ibtws` | `orderid` | `realtime` | `timeseries`

# getdata

Request current Interactive Brokers data

## Syntax

```
d = getdata(ib,ibContract)
d = getdata(ib,ibContract,eventhandler)
```

## Description

`d = getdata(ib,ibContract)` requests Interactive Brokers current data over the IB Trader Workstation connection `ib` using the IB Trader Workstation `IContract` object `ibContract` to signify the instrument.

`d = getdata(ib,ibContract,eventhandler)` requests Interactive Brokers current data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

## Examples

### Request Current Data

To request Interactive Brokers current data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Real-Time Data”. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request current data using `ib` and `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 these fields:

- Last price
- Last size
- Volume
- Bid price
- Bid size
- Ask price
- Ask size

Display the data in the `BID_PRICE` field of `d`.

```
d.BID_PRICE
ans =
  6.84
```

Close the IB Trader Workstation connection.

```
close(ib)
```

### Request Current Data Using an Event Handler

To request Interactive Brokers current data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Real-Time Data”. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request current data using `ib`, `ibContract`, and sample event handler function `ibExampleEventHandler`. Use `ibExampleEventHandler` or write a custom event handler function.

```
d = getdata(ib,ibContract,@ibExampleEventHandler)
d =
    1418.00
Columns 1 through 5
    [1x1 COM.TWS_TwsCtrl_1]    [2.00]    [1418.00]    [0]    [5.00]
Columns 6 through 7
    [1x1 struct]    'tickSize'
    ...
```

`d` is the request identifier.

After `d`, `ibExampleEventHandler` streams current data to the Command Window. Each column set is a type of tick.

For a size tick, the columns are:

- Interactive Brokers ActiveX object
- Event identifier
- Request identifier
- Tick type
- Size
- Structure that repeats the contents of the columns
- Event type

Close the IB Trader Workstation connection.

```
close(ib)
```

- “Request Interactive Brokers Real-Time 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

IBContract object

IB Trader Workstation contract, specified as an IB Trader Workstation `IBContract` object. This object is the instrument or security used in the order transaction. Create this object by calling the Interactive Brokers API function `createContract`. For details about `createContract` and the attributes that you can set, see Interactive Brokers API Reference Guide.

### **eventhandler** — Event handler

function handle | string

Event handler, specified as a function handle or a string to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

## Output Arguments

### **d** — Interactive Brokers current data

structure | double

Interactive Brokers current data, returned as a structure containing these tick types:

- Last price
- Last size
- Volume
- Bid price
- Bid size
- Ask price
- Ask size

When using an event handler function, `d` is a double denoting the request identifier.

# More About

## Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:
  - Connection
  - Information resulting from executing functions
  - Errors
- “Workflow for Interactive Brokers”
- “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20
- Interactive Brokers API Reference Guide

## See Also

`close` | `createOrder` | `history` | `ibtws` | `realtime` | `timeseries`



# history

Request Interactive Brokers historical data

## Syntax

```
d = history(ib,ibContract,startdate,enddate)
d = history(ib,ibContract,startdate,enddate,ticktype,period)
d = history(ib,ibContract,startdate,enddate,ticktype,period,
tradehours)
d = history(ib,ibContract,startdate,enddate,ticktype,period,
tradehours,eventhandler)
```

## Description

`d = history(ib,ibContract,startdate,enddate)` requests Interactive Brokers historical data using the IB Trader Workstation connection `ib` and IB Trader Workstation `IContract` object `ibContract` to signify the instrument. `history` requests data from `startdate` through `enddate`. The default tick type is 'TRADES' and default period is '1 day'.

`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`.

`d = history(ib,ibContract,startdate,enddate,ticktype,period,tradehours)` requests Interactive Brokers historical data using the flag `tradehours` to include all data or only data within regular trading hours.

`d = history(ib,ibContract,startdate,enddate,ticktype,period,tradehours,eventhandler)` requests Interactive Brokers historical data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

## Examples

### Request Interactive Brokers Historical Data with TRADES Default Tick Type and 1-Day Default Period

To request historical data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Historical Data”. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request the last 5 days of historical data using `ib` and `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    0.0079    0
    7.3534    0.0078    0.0080    0.0078    0.0079    0.1669    0.1075    0.0079    0
    7.3534    0.0079    0.0079    0.0078    0.0078    0.1982    0.1420    0.0078    0
    7.3534    0.0079    0.0080    0.0076    0.0078    0.3188    0.2239    0.0077    0
    7.3534    0.0078    0.0080    0.0077    0.0080    0.5568    0.3723    0.0079    0
```

`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:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

Display the open price for the most recent record 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 the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Historical Data”. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request the last 50 days of historical data using `ib`, `ibContract`, and these arguments:

- Start date is 50 days ago.
- End date is the current moment.
- Tick type is 'BID'.
- Bar size is '1W'.

```
startdate = floor(now)-50;
enddate = floor(now);
ticktype = 'BID';
period = '1W';
```

```
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    -0.0000    0
    7.3530    0.0080    0.0084    0.0080    0.0083    -0.0000    -0.0000    -0.0000    0
    7.3531    0.0082    0.0084    0.0081    0.0081    -0.0000    -0.0000    -0.0000    0
    7.3532    0.0080    0.0083    0.0079    0.0081    -0.0000    -0.0000    -0.0000    0
    7.3532    0.0081    0.0082    0.0079    0.0079    -0.0000    -0.0000    -0.0000    0
    7.3533    0.0079    0.0081    0.0078    0.0078    -0.0000    -0.0000    -0.0000    0
    7.3534    0.0078    0.0079    0.0077    0.0079    -0.0000    -0.0000    -0.0000    0
    7.3534    0.0079    0.0080    0.0076    0.0080    -0.0000    -0.0000    -0.0000    0
```

`d` returns the historical data for 50 days.

Each row of `d` contains historical data for 1 week.

The columns in matrix `d` are:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

Display the high price for the most recent record 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 the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Historical Data”. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request the last 50 days of historical data using `ib`, `ibContract`, and these arguments:

- Start date is 50 days ago.
- End date is the current moment.
- The empty string denotes the default tick type 'TRADES'.
- Bar size is '1M'.

```
startdate = floor(now) - 50;  
enddate = floor(now);
```

```

ticktype = '';
period = '1M';

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    0.0080    0
    7.3532    0.0080    0.0084    0.0079    0.0079    4.0250    2.6757    0.0082    0
    7.3534    0.0079    0.0081    0.0076    0.0080    3.6047    2.4843    0.0079    0

```

`d` returns the historical data for 50 days.

Each row of `d` contains historical data for 1 month.

The columns in matrix `d` are:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

Display the low price for the most recent record in matrix `d`.

```

d(1,4)

ans =
    780

```

Close the IB Trader Workstation connection.

```
close(ib)
```

### Request Interactive Brokers Historical Data Within Regular Trading Hours

To request historical data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Historical Data”. An `IContract` object is an Interactive Brokers

object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request the last 50 days of historical data using `ib`, `ibContract`, and these arguments:

- Start date is 50 days ago.
- End date is the current moment.
- The empty string denotes the default tick type 'TRADES'.
- Bar size is '1M'.
- Within regular trading hours.

```
startdate = floor(now) - 50;
enddate = floor(now);
ticktype = '';
period = '1M';
tradehours = true;
```

```
d = history(ib, ibContract, startdate, enddate, ticktype, period, ...
           tradehours)
```

d =

Columns 1 through 5

735805.00	591.25	599.55	585.21	588.85
735812.00	587.50	592.45	562.80	565.90
735819.00	568.85	575.32	560.00	568.45
...				

Columns 6 through 9

-1.00	-1.00	-1.00	0
-1.00	-1.00	-1.00	0
-1.00	-1.00	-1.00	0
...			

`d` returns the historical data for 50 days.

Each row of `d` contains historical data for 1 month.

The columns in matrix `d` are:

- Numeric representation of a date
- Open price
- High price
- Low price

- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

Display the low price for the most recent record in matrix `d`.

```
d(1,4)
```

```
ans =
    585.21
```

Close the IB Trader Workstation connection.

```
close(ib)
```

### Request Interactive Brokers Historical Data Using an Event Handler

To request historical data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Historical Data”. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request the last 50 days of historical data using `ib`, `ibContract`, and these arguments:

- Start date is 50 days ago.
- End date is the current moment.
- The empty string denotes the default tick type 'TRADES'.
- Bar size is '1M'.
- Within regular trading hours.
- Sample event handler function `ibExampleEventHandler`.

Use `ibExampleEventHandler` or write a custom event handler function.

```
startdate = floor(now) - 50;
enddate = floor(now);
ticktype = '';
period = '1M';
tradehours = true;
```

```
eventhandler = 'ibExampleEventHandler';  
  
d = history(ib,ibContract,startdate,enddate,ticktype,period,...  
           tradehours,eventhandler)  
  
d =  
    1576.00  
Columns 1 through 4  
    [1x1 COM.TWS_TwsCtrl_1]    [22.00]    [1576.00]    '20140718'  
Columns 5 through 10  
    [582.50]    [596.76]    [568.51]    [594.94]    [-1.00]    [-1.00]  
Columns 11 through 14  
    [-1.00]    [0]    [1x1 struct]    'historicalData'  
    ...
```

`d` is the request identifier.

After `d`, `ibExampleEventHandler` streams historical data to the Command Window. The columns are:

- Interactive Brokers ActiveX object
- Event identifier
- Request identifier
- Date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar
- Structure that repeats the contents of the columns
- Event type

Close the IB Trader Workstation connection.



```
close(ib)
```

- “Request Interactive Brokers Historical 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. Create this object by calling the Interactive Brokers API function `createContract`. For details about `createContract` and the attributes that 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 preceding enumerated strings. The Interactive Brokers API predetermines these strings to denote tick values to collect.

### **period** — Bar size

'1 day' (default) | '1W' | '1M'

Bar size, specified as one of the preceding enumerated strings predetermined by the Interactive Brokers API that denote the periodicity for collecting data.

### **tradehours** — Trading hours

false (default) | true

Trading hours, specified as the logical `true` or `false`. When this flag is set to `true`, this function returns data only within regular trading hours. Otherwise, this function returns all data.

Data Types: `logical`

### **eventhandler** — Event handler

function handle | string

Event handler, specified as a function handle or a string to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

## Output Arguments

### **d** — Interactive Brokers historical data

matrix | double

Interactive Brokers historical data, returned as a matrix with these columns:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume

- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

When using an event handler function, `d` is a double denoting the request identifier.

## More About

### Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:
  - Connection
  - Information resulting from executing functions
  - Errors
- “Workflow for Interactive Brokers”
- “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20
- Interactive Brokers API Reference Guide

### See Also

`close` | `createOrder` | `getdata` | `ibtws` | `realtime` | `timeseries`

# timeseries

Request Interactive Brokers aggregated intraday data

## Syntax

```
d = timeseries(ib,ibContract,startdate,enddate,barsize)
d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype)
d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype,
tradehours)
d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype,
tradehours,eventhandler)
```

## Description

`d = timeseries(ib,ibContract,startdate,enddate,barsize)` requests Interactive Brokers aggregated intraday data using the IB Trader Workstation connection `ib` and IB Trader Workstation `IContract` object `ibContract` to signify the instrument. Request data between `startdate` and `enddate` using the tick aggregation interval `barsize` for default tick type 'TRADES'.

`d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype)` requests Interactive Brokers aggregated intraday data for a specific type of market data tick `ticktype`.

`d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype,tradehours)` requests Interactive Brokers aggregated intraday data using the flag `tradehours` to include all data or only data within regular trading hours.

`d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype,tradehours,eventhandler)` requests Interactive Brokers aggregated intraday data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

## Examples

### Request Interactive Brokers Intraday Data Aggregated Every 5 Minutes with TRADES Default Tick Type

To request intraday data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Real-Time Data”. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request intraday data aggregated every 5 minutes using `ib` and `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	158.00	13.00	6.87
735329.40	6.85	6.87	6.85	6.87	29.00	24.00	6.86
735329.40	6.87	6.89	6.87	6.87	13.00	13.00	6.88
...							

`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:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

Display the open price for the most recent bar 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 the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Real-Time Data”. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request intraday data aggregated every 10 minutes using `ib`, `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   -1.00   -1.00   -1.00
    735329.17    6.38    6.38    6.38    6.38   -1.00   -1.00   -1.00
    735329.18    6.38    6.38    6.38    6.38   -1.00   -1.00   -1.00
    ...
```

`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:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price

- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

Display the high price for the most recent bar in matrix `d`.

```
d(1,3)
```

```
ans =
    6.38
```

Close the IB Trader Workstation connection.

```
close(ib)
```

### Request Interactive Brokers Intraday Data Within Regular Trading Hours

To request intraday data, set up the IB Trader Workstation connection `ib` using `ibtw`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Real-Time Data”. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request intraday data using `ib`, `ibContract`, and these arguments:

- Start date is this morning.
- End date is the current moment.
- Aggregated every 10 minutes.
- Tick type is 'BID'.
- Within regular trading hours.

```
startdate = floor(now);
enddate = now;
barsize = '10 mins';
ticktype = 'BID';
tradehours = true;

d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype,...
              tradehours)

d =
```

```
Columns 1 through 5
    735852.40    580.70    582.12    580.12    580.27
    735852.40    580.27    580.75    579.70    579.80
    735852.40    579.80    579.88    578.33    579.44
    ...

Columns 6 through 9
    -1.00    -1.00    -1.00    0
    -1.00    -1.00    -1.00    0
    -1.00    -1.00    -1.00    0
    ...
```

`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:

- Numeric representation of a date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

Display the high price for the most recent bar in matrix `d`.

```
d(1,3)
```

```
ans =
    582.12
```

Close the IB Trader Workstation connection.

```
close(ib)
```

### Request Interactive Brokers Intraday Data Using an Event Handler

To request intraday data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Real-Time Data”. An `IContract` object is an Interactive Brokers



object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request intraday data using `ib`, `ibContract`, and these arguments:

- Start date is this morning.
- End date is the current moment.
- Aggregated every 10 minutes.
- Tick type is 'BID'.
- Within regular trading hours.
- Sample event handler function `ibExampleEventHandler`.

Use `ibExampleEventHandler` or write a custom event handler function.

```

startdate = floor(now);
enddate = now;
barsize = '10 mins';
ticktype = 'BID';
tradehours = true;
eventhandler = 'ibExampleEventHandler';

d = timeseries(ib,ibContract,startdate,enddate,barsize,ticktype,...
              tradehours,eventhandler)

d =
    4853.00
Columns 1 through 3
    [1x1 COM.TWS_TwsCtrl_1]    [22.00]    [4853.00]
Columns 4 through 7
    '20140909 15:55:00'    [580.20]    [581.40]    [580.09]
Columns 8 through 13
    [581.01]    [-1.00]    [-1.00]    [-1.00]    [0]    [1x1 struct]
Column 14
    'historicalData'
...

```

`d` is the request identifier.

After `d`, `ibExampleEventHandler` streams intraday data to the Command Window. The columns are:

- Interactive Brokers ActiveX object
- Event identifier
- Request identifier
- Date
- Open price
- High price
- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar
- Structure that repeats the contents of the columns
- Event type

Close the IB Trader Workstation connection.

```
close(ib)
```

- “Request Interactive Brokers Real-Time 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. Create this object by calling the Interactive Brokers API function `createContract`. For details

about `createContract` and the attributes that 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. The Interactive Brokers API predetermines these strings to 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 preceding enumerated strings. The Interactive Brokers API predetermines these strings to denote tick values to collect.

### **tradehours** — Trading hours

false (default) | true

Trading hours, specified as the logical `true` or `false`. When this flag is set to `true`, this function returns data only within regular trading hours. Otherwise, this function returns all data.

Data Types: `logical`

### **eventhandler** — Event handler

function handle | string

Event handler, specified as a function handle or a string to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

## Output Arguments

### **d** — Interactive Brokers aggregated intraday data

matrix | double

Interactive Brokers aggregated intraday data, returned as a matrix with these columns:

- Numeric representation of a date
- Open price
- High price

- Low price
- Close price
- Volume
- Bar count
- Weighted average price
- Flag indicating if there are gaps in the bar

When using an event handler function, `d` is a double denoting the request identifier.

## More About

### Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:
  - Connection
  - Information resulting from executing functions
  - Errors
- “Workflow for Interactive Brokers”
- “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20
- Interactive Brokers API Reference Guide

### See Also

`close` | `createOrder` | `getdata` | `history` | `ibtw`s | `realtime`

# accounts

Retrieve Interactive Brokers account information

## Syntax

```
d = accounts(ib,acctno)
d = accounts(ib,acctno,eventhandler)
```

## Description

`d = accounts(ib,acctno)` retrieves account information using Interactive Brokers connection `ib` and account number `acctno`.

`d = accounts(ib,acctno,eventhandler)` retrieves account information using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

## Examples

### Retrieve Account Information

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

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

Retrieve account information for account number `acctno` using `ib`.

```
acctno = 'AB123456';
```

```
d = accounts(ib,acctno)
```

```
d =
```

```
    AccountCode: 'AB123456'
    AccountReady: 'true'
```

```
AccountType: 'LLC'
...
```

`d` is a structure with the fields containing the account information. Here, the fields are:

- Account code
- IB Trader Workstation internal use only
- Account type

For details about this data and the other fields, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(ib)
```

### Retrieve Account Information Using an Event Handler

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

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

Retrieve account information for account number `acctno` using `ib`. Use the sample event handler `ibExampleEventHandler` to display the IB Trader Workstation account information in the Command Window. Use `ibExampleEventHandler` or write a custom event handler function.

```
acctno = 'AB123456';
```

```
d = accounts(ib,acctno,@ibExampleEventHandler)
```

```
d =
```

```
 []
```

```
Columns 1 through 7
```

```
 [1x1 COM.TWS_TwsCtrl_1] [7] 'AccountCode' 'AB123456' '' 'AB123456'
```

```
Column 8
```

```
 'updateAccountValue'
```

```
 ...
```

`d` is an empty double.

The sample event handler `ibExampleEventHandler` displays the account information in the Command Window. The columns are:

- Interactive Brokers ActiveX object
- Event identifier
- Account code
- Event key
- Currency
- Account name
- Structure that repeats the contents of the columns
- Request type

For details about this data, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(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`.

### **acctno** — Account number

string

Account number, specified as a string that identifies the Interactive Brokers account number.

Example:

Data Types: char

### **eventhandler** — Event handler

function handle | string



Event handler, specified as a function handle or a string to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

## Output Arguments

### **d** — Account information

structure | double

Account information, returned as a structure containing fields with the Interactive Brokers account information. When using an event handler function, `d` is an empty double.

## More About

### Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:
  - Connection
  - Information resulting from executing functions
  - Errors
- “Workflow for Interactive Brokers”
- “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20
- Interactive Brokers API Reference Guide

### See Also

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

**Introduced in R2015a**

# contractdetails

Request Interactive Brokers contract details

## Syntax

```
[d,reqid] = contractdetails(ib,ibContract)
[d,reqid] = contractdetails(ib,ibContract,eventhandler)
```

## Description

`[d,reqid] = contractdetails(ib,ibContract)` requests Interactive Brokers contract details using IB Trader Workstation connection `ib` and IB Trader Workstation `IContract` object `ibContract`.

`[d,reqid] = contractdetails(ib,ibContract,eventhandler)` requests Interactive Brokers contract details using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

## Examples

### Request Interactive Brokers Contract Details

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

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

Create the IB Trader Workstation `IContract` object `ibContract`. Here, this object describes a security with these property values:

- Google symbol
- Stock security type
- Aggregate exchange

- USD currency

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

For details about the `IContract` object, see Interactive Brokers API Reference Guide.

Request contract details data using `ib` and `ibContract`.

```
[d,reqid] = contractdetails(ib,ibContract)
```

```
d =
```

```
    marketName: 'NMS'  
    tradingClass: 'NMS'  
    minTick: 0.01  
    ...
```

```
reqid =
```

```
    1269
```

`d` is a structure containing the contract details data including the market name, trading class name, and minimum tick. For details about this data, see Interactive Brokers API Reference Guide.

`reqid` is a number that Interactive Brokers uses to track this contract details data request.

Close the IB Trader Workstation connection.

```
close(ib)
```

### Request Interactive Brokers Contract Details Using an Event Handler

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

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

Create the IB Trader Workstation `IContract` object `ibContract`. Here, this object describes a security with these property values:

- Google symbol
- Stock security type
- Aggregate exchange
- USD currency

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

For details about the `IContract` object, see Interactive Brokers API Reference Guide.

Request contract details data using `ib`, `ibContract`, and sample event handler function `ibExampleEventHandler`. Use `ibExampleEventHandler` or write a custom event handler function.

```
[d, reqid] = contractdetails(ib, ibContract, @ibExampleEventHandler)
```

```
d =
```

```
    1269
```

```
reqid =
```

```
    1269
```

```
Columns 1 through 4
```

```
    [1x1 COM.TWS_TwsCtrl1_1]    [100]    [1269]    [1x1 Interface.Tws_ActiveX_Control_m
```

```
Columns 5 through 6
```

```
    [1x1 struct]    'contractDetailsEx'
```

`d` and `reqid` return a number that Interactive Brokers uses to track this contract details data request.

After these variables, `ibExampleEventHandler` returns contract details data to the Command Window. The columns are:

- Interactive Brokers ActiveX object

- Event identifier
- Request identifier
- Contract details ActiveX object
- Structure that repeats the contents of the columns
- Request type

For details about this data, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(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`.

### **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. Create this object by calling the Interactive Brokers API function `createContract`. For details about `createContract` and the attributes that you can set, see Interactive Brokers API Reference Guide.

### **eventhandler** — Event handler

function handle | string

Event handler, specified as a function handle or a string to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

## Output Arguments

### **d** — Interactive Brokers contract details data

structure | scalar

Interactive Brokers contract details data, returned as a structure. When using an event handler function, **d** is a scalar that denotes the contract detail data request identifier.

### **reqid** — Contract detail data request identifier

scalar

Contract detail data request identifier, returned as a scalar. Interactive Brokers uses this number to match responses to the correct data request when multiple data requests are present.

## More About

### Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:
  - Connection
  - Information resulting from executing functions
  - Errors
- “Workflow for Interactive Brokers”
- “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20
- Interactive Brokers API Reference Guide

### See Also

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

**Introduced in R2015a**

# executions

Request Interactive Brokers execution data

## Syntax

```
d = executions(ib,filter)
d = executions(ib,filter,eventhandler)
```

## Description

`d = executions(ib,filter)` requests Interactive Brokers execution data using the IB Trader Workstation connection `ib` and the Interactive Brokers execution filter `filter`.

`d = executions(ib,filter,eventhandler)` requests Interactive Brokers execution data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

## Examples

### Request Execution Filter Data

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

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

Create the IB Trader Workstation execution filter `IExecutionFilter` object `filter`. Here, this object specifies these property values:

- Buy side
- Stock security type
- Aggregate exchange
- Google symbol

```
filter = ib.Handle.createExecutionFilter;
```



```

filter.side = 'BUY';
filter.secType = 'STK';
filter.exchange = 'SMART';
filter.symbol = 'GOOG';

```

For details about the `IExecutionFilter` object, see Interactive Brokers API Reference Guide.

Request IB Trader Workstation execution filter data using `ib` and `filter`.

```

d = executions(ib,filter)
d =
    enddetails: [1x1 struct]

```

`d` is a structure containing the execution filter data in the structure `enddetails`.

Display the execution filter data.

```

d.enddetails
ans =
    Type: 'execDetailsEnd'
    Source: [1x1 COM.TWS_TwsCtrl_1]
    EventID: 38
    reqId: 1

```

The structure `enddetails` contains these fields:

- Data request type
- Interactive Brokers ActiveX object
- Event identifier
- Execution filter data request identifier

Close the IB Trader Workstation connection.

```
close(ib)
```

### Request Execution Filter Data Using an Event Handler

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

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

Create the IB Trader Workstation execution filter `IExecutionFilter` object `filter`. Here, this object specifies these property values:

- Buy side
- Stock security type
- Aggregate exchange
- Google symbol

```
filter = ib.Handle.createExecutionFilter;  
filter.side = 'BUY';  
filter.secType = 'STK';  
filter.exchange = 'SMART';  
filter.symbol = 'GOOG';
```

For details about the `IExecutionFilter` object, see Interactive Brokers API Reference Guide.

Request IB Trader Workstation execution filter data using `ib` and `filter`. Use the sample event handler `ibExampleEventHandler` to display the IB Trader Workstation execution filter data in the Command Window. Use `ibExampleEventHandler` or write a custom event handler function.

```
d = executions(ib,filter,@ibExampleEventHandler)
```

```
d =
```

```
 []
```

```
 [1x1 COM.TWS_TwsCtrl_1] [38] [1] [1x1 struct] 'execDetailsEnd'
```

`d` is an empty double.

`ibExampleEventHandler` displays the data in the Command Window. The columns are:

- Interactive Brokers ActiveX object
- Event identifier
- Execution filter data request identifier
- Structure that repeats the contents of the columns
- Data request type

For details, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(ib)
```

- “Create Interactive Brokers 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`.

### **filter** — IB Trader Workstation execution filter

`IExecutionFilter` object

IB Trader Workstation execution filter, specified as a `IExecutionFilter` object. For details about this object, see Interactive Brokers API Reference Guide.

Example:

Data Types: `struct`

### **eventhandler** — Event handler

function handle | string

Event handler, specified as a function handle or a string to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

## Output Arguments

### **d** — IB Trader Workstation execution filter data

structure | double

IB Trader Workstation execution filter data, returned as a structure. When using an event handler function, `d` is an empty double.

## More About

### Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:
  - Connection
  - Information resulting from executing functions
  - Errors
- “Workflow for Interactive Brokers”
- “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20
- Interactive Brokers API Reference Guide

### See Also

`close` | `createOrder` | `getdata` | `history` | `ibtws` | `timeseries`

**Introduced in R2015a**

# marketdepth

Request Interactive Brokers market depth data

## Syntax

```
d = marketdepth(ib,ibContract,depth)
d = marketdepth(ib,ibContract,depth,eventhandler)
```

## Description

`d = marketdepth(ib,ibContract,depth)` requests Interactive Brokers market depth data using the IB Trader Workstation connection `ib`, IB Trader Workstation `IContract` object `ibContract`, and price level `depth`.

`d = marketdepth(ib,ibContract,depth,eventhandler)` requests Interactive Brokers market depth data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

## Examples

### Request Market Depth Data

To request Interactive Brokers market depth data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Real-Time Data”. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request market depth data using `ib` and `ibContract`. Specify five price levels for the bid and offer sides for `depth`. This code assumes `ibContract` is an E-mini S&P 500 futures contract with an expiry of December 2014 that trades on the CME Globex exchange.

```
depth = 5;
d = marketdepth(ib,ibContract,depth)
d =
    bid: [5x2 double]
    offer: [5x2 double]
```

`d` is a structure that contains the fields for bid and offer price levels.

Display the bid prices for five levels of market depth.

```
d.bid
ans =
    1992.5    495
    1992.25  1479
    1992     1950
    1991.75  1763
    1991.5   2117
```

The first column contains the bid price and the second column contains the bid size.

Close the IB Trader Workstation connection.

```
close(ib)
```

### Request Market Depth Data Using an Event Handler

To request Interactive Brokers market depth data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Real-Time Data”. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Request market depth data using `ib` and `ibContract`. Specify five price levels for the bid and offer sides for `depth`. This code assumes `ibContract` is an E-mini S&P 500 futures contract with an expiry of December 2014 that trades on the CME Globex exchange. Use the sample event handler function `ibExampleEventHandler` or write a custom event handler function.

```

depth = 5;

d = marketdepth(ib,ibContract,depth,@ibExampleEventHandler)

d =

    8147
    [1x1 COM.TWS_TwsCtrl_1]    [16.00]    [8147.00]    [0]    [0]    [1.00]    [1988.7
    ...

```

`d` is the request identifier.

After `d`, `ibExampleEventHandler` streams market depth data to the Command Window.

The columns are:

- Interactive Brokers ActiveX object
- Event identifier
- Request identifier
- Position
- Operation
- Side
- Price
- Size
- Structure that repeats the contents of the columns
- Event type

Close the IB Trader Workstation connection.

```
close(ib)
```

## Input Arguments

**ib** — IB Trader Workstation connection  
connection object

IB Trader Workstation connection, specified as an IB Trader Workstation connection object created using `ibtw`.

### **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. Create this object by calling the Interactive Brokers API function `createContract`. For details about `createContract` and the attributes that you can set, see Interactive Brokers API Reference Guide.

### **depth** — IB Trader Workstation market depth

1 | 2 | 3 | ...

IB Trader Workstation market depth, specified as a scalar from one through 10. This number denotes the depth of the active book.

Data Types: `double`

### **eventhandler** — Event handler

`function handle` | `string`

Event handler, specified as a function handle or a string to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

## Output Arguments

### **d** — IB Trader Workstation market depth data

`structure` | `double`

IB Trader Workstation market depth data, returned as a structure containing the price level data for the bid and offer prices. Price level data consists of the price and size. When using an event handler function, `d` is a double denoting the request identifier.



## More About

### Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:
  - Connection
  - Information resulting from executing functions
  - Errors
- “Workflow for Interactive Brokers”
- “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20
- Interactive Brokers API Reference Guide

### See Also

`close` | `createOrder` | `history` | `ibtws` | `realtime` | `timeseries`

**Introduced in R2015a**

# orderid

Obtain next valid order identification number

## Syntax

```
id = orderid(ib)
```

## Description

`id = orderid(ib)` obtains the next valid order identification number for Interactive Brokers connection `ib`.

## Examples

### Obtain Next Valid Order Identification Number

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

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

Obtain the next valid order identification number using `ib`.

```
id = orderid(ib)
```

```
id =
```

```
54110686
```

`id` contains the next valid order identification number. Use this number in `createOrder`.

Close the IB Trader Workstation connection.

```
close(ib)
```

- “Create Interactive Brokers 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`.

## Output Arguments

### **id** — Next valid order identification number

scalar

Next valid order identification number, returned as a scalar.

## More About

### Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:
  - Connection
  - Information resulting from executing functions
  - Errors
- “Workflow for Interactive Brokers”
- Interactive Brokers API Reference Guide

### See Also

`close` | `createOrder` | `getdata` | `history` | `ibtws` | `timeseries`

Introduced in R2015a

# orders

Request Interactive Brokers open order data

## Syntax

```
o = orders(ib)
o = orders(ib,client)
o = orders(ib,client,eventhandler)
```

## Description

`o = orders(ib)` requests Interactive Brokers open order data using IB Trader Workstation connection `ib` for the current client only.

`o = orders(ib,client)` requests Interactive Brokers open order data using IB Trader Workstation connection `ib` and a client flag. `client` denotes requesting data from the current client or all clients.

`o = orders(ib,client,eventhandler)` requests Interactive Brokers open order data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

## Examples

### Request Open Order Data

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

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

Create the IB Trader Workstation `IContract` object `ibContract`. Here, this object describes a security with these property values:

- Google symbol
- Stock security type

- Aggregate exchange
- USD currency

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

Create the IB Trader Workstation `IOrder` object `ibOrder`. Here, this object describes a limit order to sell two shares with a limit price of \$590.

```
ibOrder = ib.Handle.createOrder;
ibOrder.action = 'SELL';
ibOrder.totalQuantity = 2;
ibOrder.orderType = 'LMT';
ibOrder.lmtPrice = 590;
```

For details about the `IContract` and `IOrder` objects, see Interactive Brokers API Reference Guide.

Create a unique order identifier `id`.

```
id = orderid(ib);
```

Execute the order using:

- IB Trader Workstation connection `ib`
- IB Trader Workstation `IContract` object `ibContract`
- IB Trader Workstation `IOrder` object `ibOrder`
- Unique order identifier `id`

```
d = createOrder(ib,ibContract,ibOrder,id);
```

Retrieve order information `o`.

```
o = orders(ib)
```

```
o =
```

1x2 struct array with fields:

```
Type
EventID
```

```
orderId
contract
order
orderState
```

`o` contains a structure array. The array contains a structure with data for each open order. The structure fields are:

- Order type
- Event identifier
- Order identifier
- Contract data
- Order data
- Order status

Retrieve the current status of the order.

```
o.orderState
```

```
ans =
```

```
      status: 'Submitted'
      initMargin: '1.7976931348623157E308'
      maintMargin: '1.7976931348623157E308'
      ...
```

`orderState` is a structure with fields corresponding to the status of the order. The fields are order status, initial margin, and maintenance margin. For details on these fields and the additional fields in `orderState`, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(ib)
```

### Request Open Order Data From All Clients

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

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

Create the IB Trader Workstation `IContract` object `ibContract`. Here, this object describes a security with these property values:

- Google symbol
- Stock security type
- Aggregate exchange
- USD currency

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

Create the IB Trader Workstation `IOrder` object `ibOrder`. Here, this object describes a limit order to sell two shares with a limit price of \$590.

```
ibOrder = ib.Handle.createOrder;
ibOrder.action = 'SELL';
ibOrder.totalQuantity = 2;
ibOrder.orderType = 'LMT';
ibOrder.lmtPrice = 590;
```

For details about the `IContract` and `IOrder` objects, see [Interactive Brokers API Reference Guide](#).

Create a unique order identifier `id`.

```
id = orderid(ib);
```

Execute the order using:

- IB Trader Workstation connection `ib`
- IB Trader Workstation `IContract` object `ibContract`
- IB Trader Workstation `IOrder` object `ibOrder`
- Unique order identifier `id`

```
d = createOrder(ib,ibContract,ibOrder,id);
```

Retrieve order information `o` from all clients by setting `client` to `false`.

```
o = orders(ib,false)
```

```
o =
```

```
1x2 struct array with fields:
```

```
Type
EventID
orderId
contract
order
orderState
```

`o` contains a structure array. The array contains a structure with data for each open order. The structure fields are:

- Order type
- Event identifier
- Order identifier
- Contract data
- Order data
- Order status

Retrieve the current status of the order.

```
o.orderState
```

```
ans =
```

```
        status: 'Submitted'
        initMargin: '1.7976931348623157E308'
        maintMargin: '1.7976931348623157E308'
        ...
```

`orderState` is a structure with fields corresponding to the status of the order. The fields are order status, initial margin, and maintenance margin. For details on these fields and the additional fields in `orderState`, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(ib)
```

### **Request Open Order Data Using an Event Handler**

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

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



Create the IB Trader Workstation `IContract` object `ibContract`. Here, this object describes a security with these property values:

- Google symbol
- Stock security type
- Aggregate exchange
- USD currency

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

Create the IB Trader Workstation `IOrder` object `ibOrder`. Here, this object describes a limit order to sell two shares with a limit price of \$590.

```
ibOrder = ib.Handle.createOrder;  
ibOrder.action = 'SELL';  
ibOrder.totalQuantity = 2;  
ibOrder.orderType = 'LMT';  
ibOrder.lmtPrice = 590;
```

For details about the `IContract` and `IOrder` objects, see Interactive Brokers API Reference Guide.

Create a unique order identifier `id`.

```
id = orderid(ib);
```

Execute the order using:

- IB Trader Workstation connection `ib`
- IB Trader Workstation `IContract` object `ibContract`
- IB Trader Workstation `IOrder` object `ibOrder`
- Unique order identifier `id`

```
d = createOrder(ib,ibContract,ibOrder,id);
```

Retrieve order information from all clients by setting `client` to `false` and using the sample event handler function `ibExampleEventHandler`. Use `ibExampleEventHandler` or write a custom event handler function.

```
o = orders(ib,false,@ibExampleEventHandler)
o =
    []
Columns 1 through 4
    [1x1 COM.TWS_TwsCtrl_1]    [101]    [56947638]    [1x1 Interface.Tws_ActiveX_Control]
Columns 5 through 6
    [1x1 Interface.Tws_ActiveX_Control_module.IOrder]    [1x1 Interface.Tws_ActiveX_Control_module.IOrderState]
Columns 7 through 8
    [1x1 struct]    'openOrderEx'
```

`o` contains an empty double because the event handler `ibExampleEventHandler` processes the output data.

`ibExampleEventHandler` displays the output data in the Command Window. Here, IB Trader Workstation returns:

- Interactive Brokers ActiveX object
- Event identifier
- Unique order identifier
- IB Trader Workstation `IContract` object
- IB Trader Workstation `IOrder` object
- IB Trader Workstation `IOrderState` object
- Structure that repeats the contents of the columns
- Request type

For details about this data, see [Interactive Brokers API Reference Guide](#).

Close the IB Trader Workstation connection.

```
close(ib)
```

- “Create Interactive Brokers 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`.

### **client** — Client flag

true (default) | false

Client flag, specified as a logical. `true` denotes returning data from the current client only. `false` denotes returning data from all clients.

Data Types: `logical`

### **eventhandler** — Event handler

function handle | string

Event handler, specified as a function handle or a string to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

## Output Arguments

### **o** — Interactive Brokers open order data

structure | double

Interactive Brokers open order data, returned as a structure or an empty double. The structure contains these fields:

- Order type
- Event identifier
- Order identifier
- Contract data

- Order data
- Order status

When using an event handler function, `o` is an empty double.

## More About

### Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:
  - Connection
  - Information resulting from executing functions
  - Errors
- Executing `orders` multiple times using the same IB Trader Workstation connection can cause this kind of warning message: Warning: Cannot unregister 'openOrderEx'. Invalid event name/handler combination. To fix this warning, close the IB Trader Workstation connection and create a new connection using `ibtws`.
- “Workflow for Interactive Brokers”
- “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20
- Interactive Brokers API Reference Guide

### See Also

`close` | `createOrder` | `executions` | `getdata` | `history` | `ibtws` | `orderid` | `timeseries`

**Introduced in R2015a**

# portfolio

Retrieve current Interactive Brokers portfolio data

## Syntax

```
p = portfolio(ib)
p = portfolio(ib,acctno)
p = portfolio(ib,acctno,eventhandler)
```

## Description

`p = portfolio(ib)` retrieves current Interactive Brokers portfolio data for the active account number using the IB Trader Workstation connection `ib`.

`p = portfolio(ib,acctno)` retrieves current Interactive Brokers portfolio data using the account number `acctno`.

`p = portfolio(ib,acctno,eventhandler)` retrieves current Interactive Brokers portfolio data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

## Examples

### Retrieve Current Portfolio Data

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

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

Retrieve current Interactive Brokers portfolio data using `ib`.

```
p = portfolio(ib)
```

```
p =
```

```
    Type: {5x1 cell}
    Source: {5x1 cell}
    EventID: {5x1 cell}
    contract: {5x1 cell}
    position: {5x1 cell}
    marketPrice: {5x1 cell}
    marketValue: {5x1 cell}
    averageCost: {5x1 cell}
    unrealizedPNL: {5x1 cell}
    realizedPNL: {5x1 cell}
    accountName: {5x1 cell}
```

`p` is a structure that contains these fields:

- Event type
- Interactive Brokers ActiveX object
- Event identifier
- Contract details
- Number of shares for each contract
- Price of the shares for each contract
- Number of shares multiplied by the price of the shares for each contract
- Average price when the shares are purchased for each contract
- Unrealized profit and loss for each contract
- Actual profit and loss for each contract
- Account number

`5x1` means there are five contracts in this portfolio. For details about this data, see [Interactive Brokers API Reference Guide](#).

Display the market price for each contract in the portfolio.

```
p.marketPrice
```

```
ans =
```

```
    [ 8.60]
    [582.95]
    [591.79]
    [188.44]
    [ 42.24]
```

Close the IB Trader Workstation connection.

```
close(ib)
```

### Retrieve Current Portfolio Data Using the Account Number

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

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

Retrieve current Interactive Brokers portfolio data using `ib` and account number string `acctno`.

```
acctno = 'DU111111';
```

```
p = portfolio(ib,acctno)
```

```
p =
```

```
      Type: {5x1 cell}
      Source: {5x1 cell}
      EventID: {5x1 cell}
      contract: {5x1 cell}
      position: {5x1 cell}
      marketPrice: {5x1 cell}
      marketValue: {5x1 cell}
      averageCost: {5x1 cell}
      unrealizedPNL: {5x1 cell}
      realizedPNL: {5x1 cell}
      accountName: {5x1 cell}
```

`p` is a structure that contains these fields:

- Event type
- Interactive Brokers ActiveX object
- Event identifier
- Contract details
- Number of shares for each contract
- Price of the shares for each contract
- Number of shares multiplied by the price of the shares for each contract

- Average price when the shares are purchased for each contract
- Unrealized profit and loss for each contract
- Actual profit and loss for each contract
- Account number

5x1 means there are five contracts in this portfolio. For details about this data, see [Interactive Brokers API Reference Guide](#).

Display the market price for each contract in the portfolio.

```
p.marketPrice
```

```
ans =
```

```
 [ 8.60]
 [582.95]
 [591.79]
 [188.44]
 [ 42.24]
```

Close the IB Trader Workstation connection.

```
close(ib)
```

### Retrieve Current Portfolio Data Using an Event Handler

Create the IB Trader Workstation connection `ib` on the local machine using port number 7496.

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

Retrieve current Interactive Brokers portfolio data using `ib`, account number string `acctno`, and sample event handler `ibExampleEventHandler`. Use `ibExampleEventHandler` or write a custom event handler function.

```
acctno = 'DU111111';
```

```
p = portfolio(ib,acctno,@ibExampleEventHandler)
```

```
p =
```

```
 []
```



Columns 1 through 5

```
[1x1 COM.TWS_TwsCtrl_1]    [103]    [1x1 Interface.Tws_ActiveX_Control_module.IContract
```

Columns 6 through 12

```
[515.10]    [8.22]    [21.68]    [0]    'DU111111'    [1x1 struct]    'updatePortf
...

```

`p` is an empty double because `ibExampleEventHandler` displays the current Interactive Brokers portfolio data for each security in the Command Window.

The columns are:

- Interactive Brokers ActiveX object
- Event identifier
- IB Trader Workstation `IContract` object
- Number of shares
- Price of the shares
- Number of shares multiplied by the price of the shares
- Average price when the shares are purchased
- Unrealized profit and loss
- Actual profit and loss
- Account number
- Structure that repeats the contents of the columns
- Event type

For details about this data, see Interactive Brokers API Reference Guide.

Close the IB Trader Workstation connection.

```
close(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`.

### **acctno** — Account number

string

Account number, specified as a string that identifies the Interactive Brokers account number.

Example:

Data Types: char

### **eventhandler** — Event handler

function handle | string

Event handler, specified as a function handle or a string to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20.

Example: `@eventhandler`

Data Types: `function_handle` | char

## Output Arguments

### **p** — Interactive Brokers portfolio data

structure | double

Interactive Brokers portfolio data, returned as a structure. The structure contains these fields. When using an event handler function, `p` is an empty double.

Field	Description
Type	Interactive Brokers event type name
Source	Interactive Brokers ActiveX object
EventID	Number that identifies the event type
contract	Structure that contains details for each contract in the portfolio

Field	Description
position	Number of shares for each contract in the portfolio
marketPrice	Price of the shares for each contract in the portfolio
marketValue	Number of shares multiplied by the price of the shares for each contract in the portfolio
averageCost	Average price when the shares are purchased for each contract in the portfolio
unrealizedPNL	Unrealized profit and loss for each contract in the portfolio
realizedPNL	Actual profit and loss for each contract in the portfolio
accountName	Account number

## More About

### Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:
  - Connection
  - Information resulting from executing functions
  - Errors
- “Workflow for Interactive Brokers”
- “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20
- Interactive Brokers API Reference Guide

### See Also

`close` | `createOrder` | `executions` | `getdata` | `history` | `ibtws` | `marketdepth` | `timeseries`

**Introduced in R2015a**

# realtime

Request Interactive Brokers real-time data

## Syntax

```
tickerid = realtime(ib,ibContract,f)
tickerid = realtime(ib,ibContract,f,eventhandler)
```

## Description

`tickerid = realtime(ib,ibContract,f)` requests Interactive Brokers real-time data using IB Trader Workstation connection `ib`, IB Trader Workstation `IContract` object `ibContract`, and Interactive Brokers fields `f`.

`tickerid = realtime(ib,ibContract,f,eventhandler)` requests Interactive Brokers real-time data using an event handler function `eventhandler`. Use the sample event handler `ibExampleEventHandler` or write a custom event handler function.

## Examples

### Request Real-Time Data

To request real-time data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Real-Time Data”. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Set the Interactive Brokers field `f` to `233` to denote the tick type for `RTVolume`. `RTVolume` contains these fields:

- Last trade price
- Last trade size

- Last trade time
- Total volume
- Volume weighted average price (VWAP)
- Single trade flag

For details about RTVolume, see Interactive Brokers API Reference Guide.

Request real-time data using `ib`, `ibContract`, and `f`.

```
f = '233';  
tickerid = realtime(ib,ibContract,f)  
  
tickerid =  
  
    1
```

`tickerid` returns a number for tracking the real-time data request.

The real-time data is returned in the MATLAB workspace variable `ibBuiltInRealtimeData`.

Display this real-time data.

```
ibBuiltInRealtimeData  
  
ibBuiltInRealtimeData =  
  
    id: 1  
  BID_PRICE: 584.65  
  BID_SIZE: 1  
  ASK_PRICE: 585.80  
  ASK_SIZE: 1  
  LAST_PRICE: 585  
  LAST_SIZE: 1  
  VOLUME: 11611
```

The structure `ibBuiltInRealtimeData` contains these fields:

- Real-time request identifier
- Bid price

- Bid size
- Ask price
- Ask size
- Last price
- Last size
- Volume

The `id` field is a number that tracks the real-time data request for IB Trader Workstation `IContract` object `ibContract`. When you create multiple contracts, each real-time data display has a different value for the `id` field that corresponds to a specific contract.

Cancel the real-time market data request using `tickerid`.

```
ib.Handle.cancelMktData(tickerid)
```

Close the IB Trader Workstation connection.

```
close(ib)
```

### **Request Real-Time Data Using an Event Handler**

To request real-time data, set up the IB Trader Workstation connection `ib` using `ibtws`. Create an IB Trader Workstation `IContract` object `ibContract` as shown in “Request Interactive Brokers Real-Time Data”. An `IContract` object is an Interactive Brokers object for containing the data about a security to process transactions. For details about creating this object, see Interactive Brokers API Reference Guide.

Set the field `f` to the tick type for `RTVolume` 233. `RTVolume` contains:

- Last trade price
- Last trade size
- Last trade time
- Total volume
- Volume weighted average price (VWAP)
- Single trade flag

For details about `RTVolume`, see Interactive Brokers API Reference Guide.

```
f = '233';
```

Request real-time data using `ib`, `ibContract`, and `f`. Use the sample event handler `ibExampleEventHandler` to display the real-time data in the Command Window.

```
tickerid = realtime(ib,ibContract,f,...  
                  @ibExampleEventHandler)
```

```
tickerid =
```

```
    1
```

```
    [1x1 COM.TWS_TwsCtrl_1]    [1]    [1]    [1]    [585.50]    [1]    [1x1 struct]
```

```
    [1x1 COM.TWS_TwsCtrl_1]    [2]    [1]    [0]    [1]    [1x1 struct]    'tickSize'
```

```
    ...
```

`tickerid` returns a number for tracking the real-time data request.

After the `tickerid`, `ibExampleEventHandler` streams real-time data to the Command Window. Each line is a type of tick. Here, there is a price tick and size tick.

For a price tick, the IB Trader Workstation returns:

- Interactive Brokers ActiveX object
- Event identifier
- Request identifier
- Tick type
- Price
- Automatic execution flag
- Structure that repeats the contents of the columns
- Event type

For details about this data, see Interactive Brokers API Reference Guide.

Cancel the real-time market data request using `tickerid`.

```
ib.Handle.cancelMktData(tickerid)
```

Close the IB Trader Workstation connection.



`close(ib)`

- “Request Interactive Brokers Real-Time 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 | cell array

IB Trader Workstation contract, specified as an IB Trader Workstation `IContract` object or a cell array for multiple IB Trader Workstation `IContract` objects. This object is the instrument or security used in the order transaction. Create this object by calling the Interactive Brokers API function `createContract`. For details about `createContract` and the attributes that you can set, see Interactive Brokers API Reference Guide.

Data Types: `cell`

### **f** — Interactive Brokers fields

string | cell array

Interactive Brokers fields, specified as a string or a cell array of strings. These fields correspond to numeric identifiers that specify the Interactive Brokers generic market data tick types. For details, see Interactive Brokers API Reference Guide.

Data Types: `char` | `cell`

### **eventhandler** — Event handler

function handle | string

Event handler, specified as a function handle or a string to identify an event handler function that processes the returned data. Use the sample event handler or write a custom event handler function. For details, see “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20.

Example: `@eventhandler`

Data Types: `function_handle` | `char`

## Output Arguments

### **tickerid** — Interactive Brokers market request identifier

`double`

Interactive Brokers market request identifier, specified as a `double` for tracking and canceling the market data request. `tickerid` is a scalar for one Interactive Brokers contract and a vector of scalars for multiple contracts.

## More About

### Tips

- `ibBuiltInErrMsg` appears in the MATLAB workspace. Check the status of connection and function execution by displaying the contents of this variable. `ibBuiltInErrMsg` contains messages related to:
  - Connection
  - Information resulting from executing functions
  - Errors
- “Workflow for Interactive Brokers”
- “Writing and Running Custom Event Handler Functions with Interactive Brokers” on page 1-20
- Interactive Brokers API Reference Guide

### See Also

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

**Introduced in R2015a**