



OPEN API

Core developer guide

Version: 1.6
Last Updated: 8/30/2016

Bloomberg
FOR ENTERPRISE



RELATED DOCUMENTS

DOCUMENT NAME
Core User Guide
Enterprise User Guide
Enterprise Developer Guide
Publishing User Guide
Publishing Developer Guide
Reference Guide — Bloomberg Services and Schemas

All materials including all software, equipment and documentation made available by Bloomberg are for informational purposes only. Bloomberg and its affiliates make no guarantee as to the adequacy, correctness or completeness of, and do not make any representation or warranty (whether express or implied) or accept any liability with respect to, these materials. No right, title or interest is granted in or to these materials and you agree at all times to treat these materials in a confidential manner. All materials and services provided to you by Bloomberg are governed by the terms of any applicable Bloomberg Agreement(s).

Contents

- Contents 3
- 1. About This Guide..... 6
 - 1.1. Overview 6
 - 1.1.1. API Features 6
 - 1.1.2. Bloomberg Product Features 7
- 2. Subscription Overview 9
 - 2.1. Subscription Life Cycle..... 9
 - 2.2. Starting a Subscription 10
 - 2.3. Building a Subscription..... 13
 - 2.4. Example of Building a Subscription..... 13
 - 2.5. Subscription Method 15
 - 2.6. Subscription Status Messages..... 15
 - 2.7. Subscription Data Messages 18
 - 2.8. Subscription Errors/Exceptions 21
 - 2.9. Modifying an Existing Subscription 22
 - 2.10. Stopping a Subscription 23
 - 2.11. Overlapping Subscriptions 23
 - 2.12. Receiving Data from a Subscription..... 24
 - 2.13. Snapshot Requests for Subscriptions 25
- 3. Subscription Classes..... 25
 - 3.1. SessionOptions Class 26
 - 3.2. Session Class..... 26
 - 3.3. Establishing a Connection..... 27
- 4. Data Requests..... 29
- 5. Bloomberg Services 30
 - 5.1. Service Schemas 30
 - 5.2. Accessing a Service 31
 - 5.3. Market Data Service..... 32
 - 5.4. Reference Data Service 34
 - 5.4.1. Requesting Reference Data..... 34
 - 5.4.2. Handling Reference Data Messages 35
 - 5.4.3. Handling Reference Data (Bulk) Messages 37
 - 5.5. Source Reference Service 38
 - 5.6. Custom VWAP Service 40
 - 5.7. Market Depth Data Service 40

- 5.8. Market Bar Service..... 41
- 5.9. Market List (Mktlist) Service 42
- 5.10. API Field Service (APIflds)..... 45
- 5.11. API Field Service — Field List..... 45
- 5.12. API Field Service — Field Information 46
- 5.13. API Field Service — Field Search..... 47
- 5.14. API Field Service — Categorized Field Search 48
- 5.15. Instruments..... 49
- 5.16. Page Data 49
- 5.17. Technical Analysis..... 50
- 5.18. Historical End-of-Day Study Request..... 51
- 5.19. Intraday Bar Study Request 53
- 5.20. Real-time Study Request 55
- 6. Best Practices 56
 - 6.1. Name Class..... 56
 - 6.2. Session Object 56
 - 6.3. EventQueue 56
 - 6.4. Getting Values from Elements 57
 - 6.5. MaxEventQueueSize 57
 - 6.6. Message Iterator 58
 - 6.7. CorrelationID Object..... 58
 - 6.8. DateTime..... 58
 - 6.9. If Chains 59
 - 6.10. Invariants: Un-Optimized..... 60
 - 6.10.1. Invariants: Corrected..... 61
- 7. Event Handling 62
 - 7.1. Asynchronous vs. Synchronous..... 63
 - 7.2. Multiple Sessions 63
- 8. Message Types 64
- 9. Error Codes 65
 - 9.1. Common Error Message Codes..... 65
 - 9.2. Message Return Codes (v3 API ONLY)..... 65
 - 9.2.1. General..... 65
 - 9.2.2. API Error Code number..... 67
 - 9.2.3. //BLP/APIAUTH..... 67
 - 9.2.4. //BLP/MKTDATA and //BLP/MKTVWAP 69
 - 9.2.5. //BLP/REFDATA..... 70
- 10. Event Types 72

10.1. Message Types 72

11. Handling Reference Errors/Exceptions 74

12. SDK for BLPAPI 78

13. Request/Response..... 80

 13.1. Requesting Historical Data..... 80

 13.2. Handling Historical Data Messages 81

 13.3. Requesting Intraday Bar Data 82

 13.4. Handling Intraday Bar Data Messages 83

 13.5. Requesting Intraday Tick Data 85

 13.6. Handling Intraday Tick Data Messages 87

 13.7. sendRequest Method 88

14. Multi-Threading and the API 89

 14.1. Combining Reference and Subscription Data 89

15. Converting from Excel Formulas (TransExcelFormToCOM) 90

 15.1. BDP(): Streaming Data (Real-Time or Delayed)..... 90

 15.2. BDP(): Reference Data (Static) 90

 15.3. BDS(): Bulk Data (Static) 91

 15.4. BDH(): Historical “End-of-Day” Data (Static)..... 92

 15.5. BDH(): Intraday Ticks (Static) 95

 15.6. BDH()/BRB(): Intraday Bar Data (Static/Subscription) 98

 15.7. BEQS(): Bloomberg Equity Screening 101

16. Troubleshooting..... 104

 16.1. Troubleshooting Scenarios 105

 16.2. BLP API Logging 105

1. About This Guide

The Core API “Developer’s Guide” is the starting point for learning the core usage of the Bloomberg L.P. API libraries. This knowledge will form the basis for developing applications for the **Desktop API**, **Server API**, **B-PIPE** and **Platform** products.

If the goal is to develop applications for one of the Bloomberg Enterprise products (including Server API, B-PIPE or Platform), then, upon completion of this core guide, the user will want to continue with the Enterprise User and Developer’s guides.

1.1. OVERVIEW

All API products share the same programming interface and behave almost identically. The main difference is that customer applications using the enterprise API products (which exclude the Desktop API) have some additional responsibilities, such as performing authentication, authorization and permissioning before distributing/receiving data.

1.1.1. API FEATURES

LANGUAGES

The Bloomberg v3 API is available in many popular programming and scripting languages, including Java, C/C++, .NET, Perl and Python. Bloomberg also provides a COM Data Control interface for development within Excel. The Java, .NET and C++ object models are identical, while the C interface provides a C-style version of the object model.

The Java, C and .NET API are written completely native, with .NET being written natively in C#. The C++, Python, Perl and COM Data Control interfaces are built on top of the C API libraries. Applications can be effortlessly ported among these languages as application needs change.

LIGHTWEIGHT INTERFACES

The API v3 programming interface implementations are very lightweight. The design makes the process of receiving data from Bloomberg and delivering it to applications as efficient as possible. It is possible to get maximum performance from the available versions of the interface.

EXTENSIBLE SERVICE-ORIENTED DATA MODEL

The API generically understands the concepts of subscription and request-response services. The subscribe and request methods allow the sending of requests to different data services with potentially different or overlapping data dictionaries and different response schemas at runtime. Thus the Bloomberg API can support additional services without additions to the interface—simplifying writing applications that can adapt to changes in services or entirely new services.

SUMMARY EVENTS

When subscribing to market data for a security, the API performs two actions:

1. Retrieves and delivers a summary of the current state of the security—made up of elements known as “fields.”
2. Streams all market data updates as they occur and continues to do so until subscription cancellation.

REQUEST SIZE RESTRICTIONS

Limitations exist on the number of fields for reference and historical data request: 400 fields for reference data request and 25 fields for historical data request. There is also a limit on the number of securities enforced by the Session’s MaxPendingRequests. API will split the securities in the request into groups of 10 securities and fields into groups of 128 fields. Therefore, depending of the number of securities and fields provided, the number of requests may exceed the default 1,024 MaxPendingRequests limit.

CANONICAL DATA FORMAT

Each data field returned to an application via the API is accompanied by an in-memory dictionary element that indicates the data type (e.g., double) and provides a description of the field; the data is self-describing. Data elements may be simple or complex, e.g., bulk fields. All data is represented in the same canonical form.

THREAD-SAFETY

The interface is thread safe and thread aware, giving applications the ability to utilize multiple processors efficiently.

32- AND 64-BIT PROGRAMMING SUPPORT

The Java and .NET API work on both 32- and 64-bit platforms, while the C/C++ API libraries are currently available in both 32- and 64-bit versions.

PURE JAVA IMPLEMENTATION

The Java API is implemented entirely in Java. Bloomberg did not use JNI to wrap either the existing C library or the new C++ library.

FULLY INTROSPECTIVE DATA MODEL

An application can discover a service and its attributes at runtime.

BANDWIDTH OPTIMIZED

The Bloomberg API automatically breaks large results into smaller chunks and can provide conflated streaming data to improve the bandwidth usage and the latency of applications.

THE THREE PARADIGMS

Before exploring the details for requesting and receiving data, the following three paradigms used by the Bloomberg API are described:

- Request/Response
- Subscription
- Publishing

1.1.2. BLOOMBERG PRODUCT FEATURES

FIELD-LEVEL SUBSCRIPTIONS

Updates can be requested for only the fields of interest to the application rather than receiving all trade and quote fields when a subscription is established. This reduces the overhead of processing unwanted data within both the API and the application and also reduces network bandwidth consumption between Bloomberg and its customers. Depending on a user's product, additional related fields may be included for API efficiencies, but only the fields requested should be relied upon.

INTERVAL-BASED SUBSCRIPTIONS

Many users of API data are interested in subscribing to large sets of streaming data, but only need summaries of each requested security to be delivered at periodic intervals.

24X7 ACCESS TO BLOOMBERG DATA

Provides developers with 24x7 programmatic access to data from the Bloomberg Data Center to be used in customer applications.

SECURITY LOOKUP FUNCTIONALITY

Perform a security, curve and government lookup request similar to what can be accomplished using {**SECF <GO>**}.

API DATA DICTIONARY ACCESS

Query the API Data Dictionary of available fields, providing functionality similar to what can be accomplished using {**FLDS <GO>**}.

COMMON API CONCEPTS

The following concepts must be understood before successfully requesting or subscribing to Bloomberg financial data via the API:

- Securities
- Fields/Overrides (with interactive demo)
- Historical Dates

THE THREE PARADIGMS

Request/Response

Data is requested by issuing a Request and is returned in a Message sequence consisting of zero or more PARTIAL_RESPONSE Message Events followed by exactly one RESPONSE Message Event. The final RESPONSE indicates that the Request has been completed. In general, applications using this paradigm will perform extra processing after receiving the final RESPONSE from a Request, such as performing calculations, updating a display with all data at one time stored in a cache, etc.

Subscription

A Subscription is created that results in a stream of updates being delivered in SUBSCRIPTION_DATA Message Events until the Subscription is explicitly cancelled (unsubscribed) by the application. By default, all data is returned. It is possible to set an interval where if any of the fields have changed since the last interval, a new Message is sent containing the fields that changed. Processing of interval data is the same as processing non-interval updates.

Publishing

The Bloomberg API allows customer applications to publish page-based and record-based data as well as consume it. Customer data can be published for distribution within the customer's enterprise, contributed to the Bloomberg infrastructure, distributed to others or used for warehousing. This is done via the suitable Bloomberg Platform product. Publishing applications might simply broadcast data or they can be "interactive," responding to feedback from the infrastructure about the currently active subscriptions of data consumers. Contributions and local publishing/subscribing via the Bloomberg Platform are outside the scope of this course and will be discussed in their own course.

In the case of the first two paradigms, the service usually defines which paradigm is used to access it. For example, the streaming real-time market data service (`//blp/mktdata`) uses the Subscription paradigm, whereas the reference data service (`//blp/refdata`) uses the Request/Response paradigm.

☞ **Note:** Applications that make heavy use of real-time market data should use the streaming real-time market data service. However, real-time information is available through the reference data service requests, which include the current value in the response.

2. Subscription Overview

Subscriptions are ideal for data that changes frequently and/or at unpredictable intervals. Instead of repeatedly polling for the current value, the application gets the latest value as soon as it is available without spending time and bandwidth if there have been no changes. This module contains more details on how to start, modify and stop subscriptions as well as what to expect as the result of a subscription and how to handle those results.

Currently, the subscription-based Bloomberg API core services are market data (`//blp/mktdata`), market bar (`//blp/mktbar`) and Custom VWAP (`//blp/mktwap`). Some subscription-based services are only available for B-PIPE users, such as market depth, market list and source reference data. In the future, the Bloomberg API may support delivering information other than market data through a subscription service.

This section offers a demonstration of how to subscribe to streaming data and handle its data responses.

Following are the basic steps to subscribe to market data:

1. Establish a connection to the product's communication server using the `Session.Start` method.
2. Open the `//blp/mktdata` service using the `Session.OpenService` method.
3. Build the subscription. For C/C++ API or Java API: Create a `SubscriptionList` object, which is a list of subscriptions used when subscribing and unsubscribing. Using its `add` method, add one security, along with one or more fields, desired options (e.g., `Interval`, `Delayed`) and a correlation identifier. For .NET API: Declare and instantiate a `Generic List` variable of `Subscriptions` and then add a new `Subscription` object to it for each subscription.
4. Submit the `SubscriptionList` or `Generic List` variable using the `Session.subscribe` method.
5. Handle the data and status responses, either synchronously or asynchronously, which are then returned as `Events`. An `Event` containing data will be identified by an `Event` type of `SUBSCRIPTION_DATA`, while a status `Message` will be of type `SUBSCRIPTION_STATUS`.

2.1. SUBSCRIPTION LIFE CYCLE

The life cycle of a subscription has several key points:

- **Startup:** Subscriptions are started by the `subscribe` method of `Session`. An `Event` object is generated to report the successful creation of any subscriptions and separate `Events` for each failure, if any.
- **Data Delivery:** Data is delivered in `Event` objects of type `SUBSCRIPTION_DATA`. Each `Event` has one or more `Messages` and each `Message` object has one or more correlation IDs to identify the associated subscriptions. Since each `Message` object may contain more data than requested in any individual subscription (with the exception of B-PIPE customers, who have had field filtering enabled and will, therefore, only receive the fields they subscribed), the code managing each subscription must be prepared to extract its data of interest from the `Message` object. Note: Customer applications must not rely on the delivery of data that was not explicitly requested in the subscription.
- **Modification:** A list of subscriptions (each identified by its correlation ID) can be modified by the `resubscribe` method of the `Session`.
- **Cancellation:** Subscriptions (each identified by its correlation ID) can be cancelled by the `unsubscribe` method of the `Session`.
- **Failure:** A subscription failure (e.g., a server-side failure) is indicated by an `Event` of type `SUBSCRIPTION_STATUS` containing a `Message` to describe the problem.

2.2. STARTING A SUBSCRIPTION

There are four parts to creating a subscription; however, several have default values:

- The **service name** (for example, “//blp/mktdata”). If users do not specify the service name, the `defaultSubscriptionService` of the `SessionOptions` object is used.
 - The **topic**. In the case of “//blp/mktdata,” the topic value consists of an optional symbology identifier followed by an instrument identifier. For example, “/cusip/ 097023105” and “/sedol1/2108601” include the symbology identifier, whereas “IBM US Equity” omits the symbology identifier. If users do not specify the symbology identifier, then the `defaultTopicPrefix` of the `SessionOptions` object is used.
- ☞ **Note:** *The topic’s form may be different for different subscription services.*
- The **options**. These are qualifiers that can affect the content delivered. Examples in “//blp/mktdata” include specifying an interval for conflated data.
 - The **correlation ID**. Data for each subscription is tagged with a correlation ID (represented as a `CorrelationID` object) that must be unique to the `Session`. The customer application can specify that value when the subscription is created. If the customer application does not specify a correlation ID, the Bloomberg infrastructure will supply a suitable value; however, in practice, the internally generated correlation ID is rarely used. Most customer applications assign meaningful correlation IDs that allow the mapping of incoming data to the originating request or subscription.

A user can represent any subscription as a single string that includes the service name, topic and options. For example:

- “//blp/mktdata/cusip/097023105?fields=LAST_PRICE, LAST_TRADE_ACTUAL” represents a subscription using the market data service to an instrument (BA) specified by CUSIP where any changes to the fields `LAST_PRICE` or `LAST_TRADE_ACTUAL` from the Bloomberg data model should generate an update.
- “IBM US Equity?fields=BID,ASK&interval=2” represents a subscription using the market data service to an instrument (IBM) specified by Bloomberg Ticker where any changes to the fields `BID` or `ASK` from the Bloomberg data model should generate an update subject to conflation restriction of at least two seconds between updates. In this case, the assumption is that the `Session` has a `defaultSubscriptionService` of “//blp/mktdata” and a `defaultTopicPrefix` of “ticker/”.

The Bloomberg API provides methods that accept the subscription specification as a single string as well as methods that specify the different elements of the subscription as separate parameters. Subscriptions are typically manipulated in groups, so the Bloomberg API provides methods that operate on a list of subscriptions. This example shows subscription creation by several of these methods.

```

.....
SubscriptionList subscriptions      = new
SubscriptionList(); CorrelationID   subscriptionID_IBM =
new CorrelationID(10); subscriptions.add(new
Subscription("IBM US Equity",
                                     "LAST_TRADE",
                                     subscriptionID_IBM));
subscriptions.add(new Subscription("/ticker/GOOG US Equity",
                                     "BID,ASK, LAST_PRICE",
                                     new CorrelationID(20));
subscriptions.add(new Subscription("MSFT US Equity",
                                     "LAST_PRICE",
                                     "interval=.5",
                                     new CorrelationID(30));

subscriptions.add(new Subscription(
    "/cusip/097023105?fields=LAST_PRICE&interval=5.0", //BA US
    Equity new CorrelationID(40));

session.subscribe(subscriptions);

.....

```

☞ **NOTE:** *SubscriptionList* in C# is simply an alias to *System.Collections.Generic.List<BloombergIp.Blpapi.Subscription>*, created with:

```

using SubscriptionList =
    System.Collections.Generic.List<BloombergIp.Blpapi.Subscripti
on>; SubscriptionList sl = new SubscriptionList();
sl.Add(new Subscription("4444 US Equity"));

```

Subscribing to this list of subscriptions returns an Event of type `SUBSCRIPTION_STATUS` consisting of a Message object of type `SubscriptionStarted` for each `CorrelationID`. For example, the user-defined “dump” method used in previous examples shows:

```

eventType=SUBSCRIPTION_STATUS messageType=SubscriptionStarted
CorrelationID=User: 10 SubscriptionStarted = {
}
messageType=SubscriptionStarted CorrelationID=User: 20 SubscriptionStarted =
{
}
messageType=SubscriptionStarted CorrelationID=User: 30 SubscriptionStarted =
{
}
messageType=SubscriptionStarted CorrelationID=User: 40 SubscriptionStarted =
{
}

```

In case of an error, an Event is available to report the subscriptions that failed. For example, if the specification for MSFT (CorrelationID 30) above was mistyped (MSFTT), that would result in the following Event:

```

eventType=SUBSCRIPTION_STATUS
messageType=SubscriptionFailure
CorrelationID=User: 30
SubscriptionFailure = {
    reason = {
        source =
        BBDB@p111
        errorCode = 2
        category =
        BAD_SEC
        description = Invalid security
    }
}

```

2.3. BUILDING A SUBSCRIPTION

Several typical subscription parts have default values:

- **Service Name** (e.g., “//blp/mktdata”): If the service name is not specified, the defaultSubscriptionService of the SessionOptions object is used, which is “//blp/mktdata”.
- **Topic**: In the case of “//blp/mktdata,” the topic value consists of an optional topic prefix followed by an instrument identifier. For example, “/cusip/097023105” and “/sedol1/2108601” include the topic prefix, whereas “IBM US Equity” omits the topic prefix. If the topic prefix is not specified, the defaultTopicPrefix of the SessionOptions object is used, which is “/ticker” by default. Therefore, if using a ticker, such as IBM, the security string would be “IBM US Equity,” with the “/ticker” topic prefix implied. Note: The topic’s form may be different for different subscription services.
- **Fields**: API fields define the type of data to retrieve for the specified topic. These are required, comma-delimited and are prefixed with a question mark immediately following the topic. Make sure that the fields being used are real-time fields, indicated by white text on {FLDS <GO>} on the Bloomberg Professional@ service. Otherwise, an invalid field error within the response for that security’s subscription will be returned.
- **Options**: These are optional qualifiers that can affect the content delivered. Examples for “//blp/mktdata” include specifying an interval for conflated data (i.e., interval=n, where n is measured in seconds) or indicating receipt of delayed data, which is accomplished via the “delayed” option. They are prefixed by an ampersand (&). If using the “//blp/mktwap” service, specify one or more override field/value pairings. If using the Desktop API, the “useGMT” option can be used—that will ensure that all time data being returned is converted to GMT/UTC. This option is available only for the Desktop API because the streaming data is initially being adjusted based on the Bloomberg Professional service’s {TZDF <GO>} settings. All other products will default to receiving time-specific streaming data in GMT/UTC (unadjusted).
- **Correlation ID (optional)**: Data for each subscription is tagged with a correlation ID, represented as a CorrelationID object, that must be unique to the session. The customer application can specify that value when the subscription is created. If the customer application does not specify a correlation ID, the Bloomberg infrastructure will supply a suitable value. In practice, the internally generated correlation ID is rarely used. Most customer applications assign meaningful correlation IDs that allow the mapping of incoming data to the originating request or subscription.

2.4. EXAMPLE OF BUILDING A SUBSCRIPTION

The Bloomberg API provides class methods that accept the subscription specification as a single string as well as methods wherein the different elements of the subscription are specified as separate parameters. Subscriptions are typically manipulated in groups so Bloomberg provides methods that operate on a list of subscriptions. To start with some sample subscription strings:

“//blp/mktdata/cusip/097023105?fields=LAST_PRICE, LAST_TRADE_ACTUAL”: Represents a subscription using the market data service for an instrument (Boeing Co.) specified by CUSIP where any changes to the fields LAST_PRICE or LAST_TRADE_ACTUAL from the Bloomberg data model should generate an update.

“IBM US Equity?fields=BID, ASK&interval=2”: Represents a subscription using the market data service to an instrument (IBM) specified by a Bloomberg Ticker where any changes to the fields BID or ASK from the Bloomberg data model should generate an update subject to conflation restriction of at least two seconds between updates. In this case, based on the assumption that the Session has a defaultSubscriptionService of “//blp/mktdata” and a defaultTopicPrefix of “ticker/”.

“GOOG US Equity?fields=LAST_PRICE, BID, ASK&delayed”: Represents a delayed subscription using the market data service to an instrument (GOOG) specified by a Bloomberg Ticker where any changes to the fields LAST_PRICE, BID or ASK from the Bloomberg data model should generate a delayed update. As in the previous example, based on the assumption that the Session has a defaultSubscriptionService of “//blp/mktdata” and a defaultTopicPrefix of “ticker/”.

As an alternative to representing a subscription as a single string that includes the service name, topic, fields and options, the subscription string can be broken down into separate parameters. In the following example, subscriptions are created using both methods:

```
const char *security1 = "IBM US Equity";
const char *security2 = "/cusip/912828GM6@BGN";
SubscriptionList subscriptions;
subscriptions.add(security1,
                 "BID,ASK,LAST_PRICE",
                 "interval=3",
                 CorrelationID((char *)security2));
subscriptions.add(security2,
                 "LAST_PRICE",
                 "interval=.5",
                 CorrelationID(10));
subscriptions.add(security1,
                 "LAST_PRICE",
                 "delayed");
subscriptions.add(security1,
                 "LAST_PRICE",
                 "delayed");
subscriptions.add(security1,
                 "LAST_PRICE",
                 "delayed");
subscriptions.add("/cusip/097023105?fields=LAST_PRICE&interval=5.0",
                 //BA Equity
                 CorrelationID(30));
subscriptions.add("//blp/mktdata/isin/US0970231058?fields=LAST_PRICE&interval=5.0",
                 //BA Equity
                 CorrelationID(40));
session.subscribe(subscriptions);
```

2.5. SUBSCRIPTION METHOD

The Subscribe method is a member of the Session class and is used to submit a list of subscriptions to be filled by the Bloomberg Data Center along with a user's Identity object if required.

Add one or more subscriptions to the subscription list utilizing any of the ADD method overloads of the SubscriptionList class. The SubscriptionList object is then submitted via the Subscribe method. The two method overload definitions are:

```
void blpapi::Session::subscribe(const SubscriptionList &subList,
                               const char *requestLabel=0, int requestLabelLen=0)
void blpapi::Session::subscribe(const SubscriptionList &subList, const Identity &ID,
                               const char *requestLabel=0, int requestLabelLen=0)
```

Begin subscriptions for each entry in the specified SubscriptionList using the specified identity for authorization. Optional requestLabel and requestLabelLen define a string that will be recorded along with any diagnostics for this operation. There must be at least requestLabelLen printable characters at the location requestLabel. A SUBSCRIPTION_STATUS Event will be generated for each entry in the SubscriptionList.

Several supporting methods are available to resend, modify and cancel an existing subscription.

.NET API Developers

Unlike when working with the Java and C++ API interfaces—instead of using the subscriptionList object, instantiate a Generic List variable of Subscriptions and then add a new Subscription object to it for each subscription.

2.6. SUBSCRIPTION STATUS MESSAGES

Irrespective of handling Messages synchronously or asynchronously, the status subscription Messages that are returned are captured in the same manner. This is done by handling the Event object's SESSION_STATUS, SERVICE_STATUS, SUBSCRIPTION_STATUS, and AUTHORIZATION_STATUS messages.

A SUBSCRIPTION_STATUS Event type, for instance, may contain a SubscriptionStarted or SubscriptionFailure Message type.

Notes:

- When using an asynchronous Session, the application must be aware that because the callbacks are generated from another thread, they may be processed before the call that generates them has returned. For example, the SESSION_STATUS Event generated by a startAsync may be processed before startAsync has returned (even though startAsync itself will not block).
- Subscription data Messages are also handled in this same event handler. They were omitted in the code for simplicity.

Following is the skeleton status code placed within the Event handler. In this case, the messages are handled asynchronously inside the Session's processEvent handler:

```
<C++>
public void processEvent(Event event, Session session) {
    switch (event.eventType())
    {
        case Event::SERVICE_STATUS: {
```

```

        // If service opened successfully, send request.
        break;
    }
    case Event::SUBSCRIPTION_STATUS: {
        // If subscription fails, then perform appropriate action
        break;
    }
    case Event::AUTHORIZATION_REQUEST:
        // If authorization was successful, perform next action.
        break;
    }
    default: {
        // Handle unexpected response.
        break;
    }
}
}
}

```

Following is the processing of the Event returned for starting the Session. If successful, the code will attempt to open the needed service. Since the openServiceAsync method generates an exception on failure, but processEvent is not allowed to omit an exception, that call must be surrounded by a try-catch block. In event of failure, the example chooses to terminate the process.

```

<C++>
Case Event::SESSION_STATUS: {
    MessageIterator iter = event.messageIterator();
    while (iter.hasNext()) {
        Message message = iter.next();
        if (message.messageType().equals("SessionStarted")) {
            try {
                session.openServiceAsync("//blp/mktdata",
CorrelationID(99));
            } catch (Exception &e) {
                std::cerr << "Could not open //blp/mktdata for async";
                std::exit(-1);
            }
        }
    }
}
}

```



```

        }
    } else {
        // Handle error.
    }
}
break;
}

```

On receipt of a `SERVICE_STATUS` type Event, the Messages are searched for one indicating that the `openServiceAsync` call was successful. The Message type must be “ServiceOpened” and the correlation ID must match the value assigned when the request was sent.

If the service was successfully opened, users can create, initialize and send a request. The only difference is that the call to subscribe must be guarded against the transmission of exceptions—not a concern until now.

```

<C++>
public void processEvent(Event event, Session session) {
    switch (event.eventType())
    {
        case Event::SERVICE_STATUS: {
            // If service opened successfully, send request.
            break;
        }
        case Event::SUBSCRIPTION_STATUS: {
            // If subscription fails, then perform appropriate action
            break;
        }
        case Event::AUTHORIZATION_REQUEST:
            // If authorization was successful, perform next action.
            break;
        }
        default: {
            // Handle unexpected response.
            break;
        }
    }
}

```

```
}

```

2.7. SUBSCRIPTION DATA MESSAGES

Once a subscription has started, the application will receive updates for the requested data in the form of Messages contained within Event objects of type SUBSCRIPTION_DATA. Within each Message there is a CorrelationID to identify the subscription that requested the data.

The “//blp/mktdata” service typically responds with Messages that have more data than was requested for the subscription. In this example, only updates to the LAST_TRADE field of IBM were requested in the subscription corresponding to CorrelationID 10. Applications must be prepared to extract the data they need and to discard the rest.

Here is a sample dump of a subscription data Message:

```
eventType=SUBSCRIPTION_DATA
messageType=MarketDataEvents
CorrelationID=User: 10
MarketDataEvents = {
    LAST_PRICE = 212.64
    BID = 212.63
    ASK = 212.64
    MKTDATA_EVENT_TYPE = SUMMARY
    MKTDATA_EVENT_SUBTYPE = INITPAINT
    IS_DELAYED_STREAM = false
...

```

In the above output, the Event type is “SUBSCRIPTION_DATA”. This is the Event type handled in the code.

The next step would be taking a deeper look into a subscription-based data event handler. Displayed below is the partial code sample that handles those particular subscription Messages. The handleOtherEvent function definition has been removed for brevity.

```
<C++>
#include <blpapi_correlationid.h>
#include <blpapi_event.h>
#include <blpapi_message.h>
#include <blpapi_request.h>
#include <blpapi_session.h>
#include <blpapi_subscriptionlist.h>

```

```

#include <iostream>
using namespace BloombergLP;
using namespace blpapi;
static void handleDataEvent(const Event& event, int updateCount)
{
    std::cout << "EventType=" << event.eventType() << std::endl;
    std::cout << "updateCount = " << updateCount << std::endl;
    MessageIterator iter(event);
    while (iter.next()) {
        Message message = iter.message();
        std::cout << "correlationId = " << message.correlationId() <<
std::endl;
        std::cout << "messageType = " << message.messageType() << std::endl;
        message.print(std::cout);
    }
}
int main(int argc, char **argv)
{
    SessionOptions sessionOptions;
    sessionOptions.setServerHost("localhost");
    sessionOptions.setServerPort(8194);
    Session session(sessionOptions);
    if (!session.start()) {
        std::cerr <<"Failed to start session." << std::endl;
        return 1;
    }
    if (!session.openService("//blp/mktdata")) {
        std::cerr <<"Failed to open //blp/mktdata" << std::endl;
        return 1;
    }
    CorrelationId subscriptionId((long long)2);
    SubscriptionList subscriptions;
    subscriptions.add("AAPL US Equity", "LAST_PRICE", "", subscriptionId);
}

```

```

session.subscribe(subscriptions);
int updateCount = 0;
while (true) {
    Event event = session.nextEvent();
    switch (event.eventType()) {
        case Event::SUBSCRIPTION_DATA:
            handleDataEvent(event, updateCount++);
            break;
        default:
            handleOtherEvent(event); // This function, if shown,
                                     // would handle Status messages,
                                     // such as session terminated,
                                     // and so forth
            break;
    }
}
return 0;
}

```

In the previous code sample, it is important to observe the following:

- The tick data from the subscribe call is being handled in synchronous mode. This is determined by the combination of the nextEvent call and the fact that no EventHandler is being provided to the Session object when it is being created. Remember that if nextEvent is called when an EventHandler is provided, then an exception will be thrown.
- When the Event type is `Event::SUBSCRIPTION_DATA`, the `handleDataEvent` method is then called where a MessageIterator is created and used to iterate over the Message objects within the Event.
- The Event type and counter are then printed and, for each tick update, the correlation ID, Message type and tick data dump are also sent to `std::cout`.
- The correlation ID in this case will be the value of the `subscriptionId` variable that was sent with the subscription. In an application with more than the one security being subscribed, assign a different correlation id to each SubscriptionList item and use them to map the tick update to the actual subscription that was sent.
- Not displayed in this code snippet is the `handleOtherEvent` method code.

2.8. SUBSCRIPTION ERRORS/EXCEPTIONS

When handling the subscription status Messages, a “reason” Message type denotes a subscription failure. If an “exceptions” Message type is received, this usually means that at least one field was valid while others were invalid. In the latter case, check the number of exceptions and determine the invalid field(s) in the handler. Following is sample code that demonstrates how to handle a “reason” Message type:

```
<C++>
if (msg.hasElement("reason")) {
    Element reason = msg.getElement("reason");
    fprintf(stdout, "          %s: %s\n",
            reason.getElement("category").getValueAsString(),
            reason.getElement("description").getValueAsString());
}
```

Following is a sample printout of a “reason” Message that includes the category and description:

```
SubscriptionFailure = {
    reason = {
        category = BAD_SEC
        description = Invalid security
    }
}
```

Following is sample code that demonstrates how to handle an “exceptions” Message type:

```
<C++>
if (msg.hasElement("exceptions")) {
    Element exceptions = msg.getElement("exceptions");
    for (size_t i = 0; i < exceptions.numValues(); ++i) {
        Element exInfo = exceptions.getValueAsElement(i);
        Element fieldId = exInfo.getElement("fieldId");
        Element reason = exInfo.getElement("reason");
        fprintf(stdout, "          %s: %s\n",
            fieldId.getValueAsString(),
            reason.getElement("category").getValueAsString());
    }
}
```

2.9. MODIFYING AN EXISTING SUBSCRIPTION

Once a subscription has been created, the options can be modified (e.g., to change the fields in the subscription) using the resubscribe method of the Session.

☞ **Note:** Use of the resubscribe method is generally preferred to cancelling the subscription (unsubscribe method) and creating a new subscription because updates might be missed between the unsubscribe and subscribe calls.

The resubscribe method accepts a SubscriptionList. For example, let’s say that users initially subscribed to “IBM US Equity” along with LAST_PRICE and BID, passing subscriptionID_IBM as the correlation identifier. This would look like the following:

```
SubscriptionList subscriptions = new SubscriptionList;
subscriptions.add("IBM US Equity", "LAST_PRICE, BID", subscriptionID_IBM);
session.subscribe(subscriptions);
```

If users would like to modify that subscription and remove the LAST_PRICE field, retain the BID field and add the ASK field, the following code fragment can be used:

```
SubscriptionList subscriptions = new SubscriptionList;
subscriptions.add("IBM US Equity", "BID,ASK", subscriptionID_IBM);
session.resubscribe(subscriptions);
```

In the above code snippet, BID was included in the initial subscription, so it will continue being subscribed, while LAST_PRICE will be unsubscribed and ASK will be subscribed. This resubscribed list will simply replace the initial one made under the same correlation identifier.

☞ **Note:** *The client receives an Event object indicating successful resubscription (or not) before receipt of any data from that subscription. Also, the behavior is undefined if the topic of the subscription (e.g., the security itself) is changed.*

2.10. STOPPING A SUBSCRIPTION

The Bloomberg API provides a cancel method that will cancel a single subscription (specified by its CorrelationID) and another method (unsubscribe) that will cancel a list of subscriptions. The following code fragment utilizes the unsubscribe method to cancel all of the subscriptions created earlier:

```
SubscriptionList subscriptions = new SubscriptionList;
for (int id = 10; id <= 40; id += 10) {
    subscriptions.add("IBM US Equity", new CorrelationID(id));
    // Note: The topic string is ignored for unsubscribe.
}
session.unsubscribe(subscriptions);
```

☞ **Note:** *No Event is generated for unsubscribe.*

2.11. OVERLAPPING SUBSCRIPTIONS

An application may make subscriptions that “overlap.”

One form of overlap occurs when a single incoming update may be relevant to more than one subscription. For example, two or more subscriptions may specify the updates for the same data item. This can easily happen inadvertently by “topic aliasing”, where one subscription specifies a security by ticker, the other by CUSIP.

Another form of overlap occurs when separate data items intended for different subscriptions on the customer application process arrive in the same Message object. For example, the Bloomberg infrastructure can improve performance by packaging two data items within the same Message object. This can occur when a customer’s application process has made two separate subscriptions, where one includes a request for “IBM US Equity” and “LAST_TRADE,” while the second includes “IBM US Equity” and “LAST_TRADE”.

The customer application developer can specify how the Bloomberg API should handle overlapping subscriptions. The behavior is controlled by the allowMultipleCorrelatorsPerMsg option to the SessionOptions object accepted by the Session constructor.

If the `allowMultipleCorrelatorsPerMsg` option is false (the default), then a Message object that matches more than one subscription will be returned multiple times from the MessageIterator, each time with a single, different CorrelationID.

If the `allowMultipleCorrelatorsPerMsg` object is true, then a Message object that matches more than one subscription will be returned just once from the MessageIterator. The customer application developer must supply logic to examine the multiple CorrelationID values (see the `numCorrelationIds` and `correlationIDAt` methods of the Message class) and dispatch the appropriate data to the correct application software.

2.12. RECEIVING DATA FROM A SUBSCRIPTION

Once a subscription has started, the application will receive updates for the requested data in Message objects arriving as Event objects of type `SUBSCRIPTION_DATA`. With each Message, there is a CorrelationID to identify the subscription that requested the data.

The `“//blp/mktdata”` service typically responds with Messages that have more data than was requested for the subscription. In this example, only updates to the `LAST_TRADE` field of IBM were requested in the subscription corresponding to CorrelationID 10.

Applications must be prepared to extract the data that they need and to discard the rest.

☞ *For additional information, refer to ““//blp/mktdata” service.*

```
eventType=SUBSCRIPTION_DATA
messageType=MarketDataEvents
CorrelationID=User: 10
MarketDataEvents = {

    IND_BID_FLAG = false
    IND_ASK_FLAG = false
    IS_DELAYED_STREAM = true
    TIME =
    14:34:44.000+00:00

    VOLUME = 7589155

    RT_OPEN_INTEREST = 8339549

    RT_PX_CHG_PCT_1D = -0.32

    VOLUME_TDY = 7589155
    LAST_PRICE = 118.15
    HIGH = 118.7
    LOW = 116.6
    LAST_TRADE = 118.15
    OPEN = 117.5
    PREV_SES_LAST_PRICE = 118.53
    EQY_TURNOVER_REALTIME = 8.93027456E8
    RT_PX_CHG_NET_1D = -0.379999

    OPEN_TDY = 117.5

    LAST_PRICE_TDY = 118.15

    HIGH_TDY = 118.7

    LOW_TDY = 116.6
    RT_API_MACHINE = p240
```



```

    API_MACHINE = p240
    RT_PRICING_SOURCE =
    US_EXCH_CODE_LAST = D
    EXCH_CODE_BID = 0

    SES_START = 09:30:00.000+00:00

    SES_END = 16:30:00.000+00:00

}

```

2.13. SNAPSHOT REQUESTS FOR SUBSCRIPTIONS

For applications that require data on an on-demand basis from subscription services, snapshot functionality is available. In this type of request, a subscription is created, but messages are sent to the application only when called. This reduces latency for receiving the latest data when compared to starting a new subscription each time.

A RequestTemplate is used to maintain the subscription for snapshot, which remains open and updating until the template is canceled as per any subscription. Responses to the snapshot request are the same as a regular subscription with the event type “Summary”, and the event subtype “INITPAINT”.

NOTE: This is only available in Enterprise Products

CODE EXAMPLE

A code example is available in the C++, Java, and .NET SDK as SnapshotRequestTemplateExample:

```

<C++>

//Create template for request

std::vector<RequestTemplate> snapshots;
std::string subscriptionString(d_service + d_topic + fieldsStr);
snapshots.push_back(session.createSnapshotRequestTemplate(
                                                                    subscriptionString.c_str(),
                                                                    CorrelationId(d_topic),
                                                                    subscriptionIdentity));

// Request snapshot based on a specific template
session.sendRequest(snapshots[0], correlationID(0));

```

3. Subscription Classes

Before delving deeper into the various classes and class members available via the API libraries, it is important to have a high-level understanding of the typical structure of a Bloomberg API application and how it may differ depending upon the product used. This module will describe the core components that make up a typical Bloomberg API application.

The two API classes that are responsible for both connecting users application to their Bloomberg API product's communication server process and performing the necessary authentication, authorization and data request duties are *SessionOptions* and *Session*, which will be covered next.

3.1. SESSIONOPTIONS CLASS

The SessionOptions class is used to create well-defined defaults for various options. Following is a list of a few of the important methods/properties of this class. Note that there may be slight differences in the names of some of the class members depending upon the API language interface used:

SERVERHOST

Read/write property representing the host-name or the IP address of where the communication process is running. The default is “localhost”.

SERVERPORT

Read/write property representing the port number of the communication process. The default is 8194.

DEFAULTSUBSCRIPTIONSERVICE

Read/write property representing the name of the service to be used by default for subscriptions. The default is “//blp/mktdata”.

DEFAULTTOPICPREFIX

Read/write property representing the name of the topic prefix to be used for subscriptions if one is not provided. The default is “ticker”.

AUTORESTARTONDISCONNECTION

Read/write property that accepts a Boolean value and determines whether the session is automatically restarted if there is a connection failure. Set the NumStartAttempts property in tandem with this property.

ALLOWMULTIPLECORRELATORSPERMSG

Read/write property that accepts a Boolean value and allows a data response comprising multiple correlators.

SERVERADDRESSES (SERVER API, B-PIPE, PLATFORM)

Read/write property used in tandem with the ServerAddress class; used to set one or more port and host pairings that will be used in order if the connection is lost to the primary host/port.

AUTHENTICATIONOPTIONS (B-PIPE)

Method used for both user and application authentication. In this method, pass a string, which specifies the AuthenticationType, for a user (choices are OS_LOGON and DIRECTORY_SERVICE) and AuthenticationMode and ApplicationAuthenticationType parameters with values of APPLICATION_ONLY and APPNAME_AND_KEY, respectively, for an application.

The authentication and permissioning systems of Server API and B-PIPE require use of the //blp/apiauth service. This defines the requests and responses that will come from the API.

The SessionOptions class is used to set the Session parameters prior to making the connection to the Bloomberg Communication Server (bbcomm, Server API, B-PIPE) process. The SessionOptions class is then used as an argument when creating an instance of the Session object.

Note for Desktop API Developers: Desktop API developers will rarely need to use the SessionOptions object in their application as they always connect to BBComm, which runs on the same PC as the Bloomberg Professional service (localhost) and usually connects on port 8194, which are the defaults of the ServerHost and ServerPort properties, respectively.

3.2. SESSION CLASS

The Session class provides a session for establishing a connection along with data-request–related tasks. Following are important constructor overloads:

- Session(SessionOptions sessionOptions): Creates a Session with the specified Session Options
- Session(SessionOptions sessionOptions, EventHandler handler): Creates a Session with the specified Session Options and dispatches Events on this Session to the specified handler.

Important methods of the Session class that are used by all API products include:

START

This establishes a connection with the product's communication server process. It issues a blocking call to start the Session.

STOP

This stops the operation of the Session.

OPENSERVICE

It opens the service having the specified URI (e.g., `“//blp/refdata”`, `“//blp/mktdata”`, etc).

GETSERVICE

This service returns a handle to a `BloombergBlpapi.Service` object representing the service identified by the specified URI. This method is not used for a subscription-based request (e.g., `“//blp/mktdata”`).

SENDREQUEST

This sends a completed request to the service, including `“//blp/refdata”`, `“//blp/apiflds”` and `“//blp/tasvc”`.

SUBSCRIBE

This sends a subscription list to be subscribed. It is used for services such as `“//blp/mktdata”`, `“//blp/mktvwap”`, and `“//blp/mktbar”`.

RESUBSCRIBE

Modifies each subscription in the specified `SubscriptionList` (for C++ API and Java) or generic `List<Subscription>` (for Java API and .NET API) to reflect the modified options specified.

UNSUBSCRIBE

Cancels previous requests for asynchronous topic updates associated with the `CorrelationIDs` listed in the specified `SubscriptionList` (for C++ API and Java) or generic `List<Subscription>` (for Java API and .NET API).

CREATESNAPSHOTREQUESTTEMPLATE

Creates template to be used for snapshot subscription requests.

CANCEL

Cancels outstanding subscriptions, templates or requests represented by the specified `CorrelationIDs` or generic `List<CorrelationID>`.

Following are a few of the important methods of the Session class needed by the enterprise API products (Server API, Platform and B-PIPE):

GENERATETOKEN

Requests a token string for the specified application and/or user entered in EMRS, which is then used for authorization.

SENDAUTHORIZATIONREQUEST

Sends a completed authorization request to the authorization service (`“//blp/apiauth”`).

CREATEIDENTITY

Returns an identity that is valid but not authorized.

3.3. ESTABLISHING A CONNECTION

In most cases, both the Session and SessionOptions classes will be used to establish a connection between an application and the communication server process associated with the product (e.g., `serverapi`, `bbcomm`, etc.). Take a look at the accompanying slide for a code sample for establishing a connection.

As mentioned earlier, a Desktop API application will always connect to `bbcomm`, which will be running on the same PC on port 8194. Since the default values of the `ServerHost` and `ServerPort` properties of the `SessionOptions` object are `“localhost”` and 8194, respectively, a Desktop API application would not be required to set them. In fact, it is not

uncommon to find a Desktop API application that does not use this class object for this reason, but may call its other class members.

This is not the case for the enterprise API products, which will require that at least one of these two properties be set or possibly not used at all. In place of those two properties, an enterprise application wishing to handle failover situations would use the `ServerAddresses` and `AutoRestartOnDisconnection` members of the `SessionOptions` class. If the connection is lost to the primary host/server, the application will connect to the next pairing specified by the `ServerAddresses` property. To use this property, the `AutoRestartOnDisconnection` property must be set to true (default is false). Failover will be covered in detail within the Enterprise documentation.

In an Enterprise application that is not handling failover, the `ServerHost` and `ServerPort` properties will be set. For instance, if the `serverapi` process was installed on a Windows, Solaris or Linux box with the defaults, it will most likely be listening on port 8194, which is the default of the `SessionOptions.ServerPort` property as previously mentioned. Since the server process is probably not installed on the same machine as the application (unless possibly testing on a development machine), the `SessionOptions.ServerHost` property will not be the default “localhost” value.

Once the required properties are set, the `SessionOptions` object will be used when instantiating the `Session` object. The settings will take effect when the `Session.Start` method is called on to establish a connection between the application and the communication server process.

4. Data Requests

STREAMING DATA (MARKET DATA)

Real-time and/or delayed streaming tick data is retrieved by subscribing to or monitoring a list of securities and real-time fields. Once a subscription is made, continuous or intervalized tick updates will be sent from the Bloomberg Data Center until the user cancels or unsubscribes from that subscription. The `Subscribe` method will be used.

PER REQUEST DATA (REFERENCE DATA / BULK DATA)

Data retrieved by sending a Request object via the `SendRequest` method. Per request data comprises of values for the requested securities/fields at that moment in time, assuming the user has real-time entitlements to that data. Otherwise, delayed values will be received. Static fields (e.g., `PX_LAST` instead of `LAST_PRICE`, where the latter is a real-time field) should be used. However, real-time fields will also work in this type of request as each has an equivalent static field.

Bulk data is also a of data, but it is represented by more than one piece of data and requires special handling. For instance, `COMPANY_ADDRESS` is a bulk field and returns multiple lines of information. Bulk fields are identified on `{FLDS <GO>}` by the “Show Bulk Data...” text in its Value column. For both types of data, a Request object of type `ReferenceDataRequest` is created.

This can be tested by loading “IBM US Equity” on the Bloomberg Professional service, running `{FLDS <GO>}` in the same window and entering “company address” in the yellow query field below the security and pressing `<GO>`.

HISTORICAL DATA

Data represented by a single point of data for each day specified in the date range of the historical request. Use the `SendRequest` method to make this request. A Request object of type `HistoricalDataRequest` will be created. Before submitting the Request object to Bloomberg, add any securities and (historical) fields, a start and end date range, and set any applicable elements to customize the request. For instance, to only receive data for trading days or data for all calendar days, fill in the non-trading days with the previous trading day’s value.

INTRADAY DATA

Data available as:

- **Tick data** — Each tick over a defined period of time
- **Bar data** — A series of intraday summaries over a defined period of time for a security and Event type pair

In both cases, pass the request using the `SendRequest` method, but only specify one security for each

Request object submitted. For intraday ticks, create an object of type `IntradayTickRequest`.” For bars, create an object of type `IntradayBarRequest`.”

5. Bloomberg Services

There are two Core service schemas and several additional services that are common to all or some of the API products.

SERVICES

1. **Market Data** — `//blp/mktdata`: Used when subscribing to streaming real-time and delayed market data.
2. **Reference Data** — `//blp/refdata`: Used when requesting reference data such as pricing, historical/time-series and intraday bars and ticks.
3. **Source Reference** — `//blp/srceref`: Used to subscribe to the source reference and tick-size data available for the specified entitlement ID.
4. **Vwap** — `//blp/mktvwap`: Subscription-based service used when requesting streaming custom Volume-Weighted-Average-Price data.
5. **Market Depth (B-PIPE ONLY)** — `//blp/mktdepthdata`: Market Depth Data service.
6. **Market Bar** — `//blp/mktbar`: Subscription-based service used when requesting streaming real-time intraday market bar data. This service is currently unavailable to B-PIPE users.
7. **Market List** — `//blp/mktlist`: Used to perform two types of list data operations. The first is to subscribe to lists of instruments, known as “chains,” using the “chain” <subservice name> (i.e., `//blp/mktlist/chain`). The second is to request a list of all the instruments that match a given topic key using the “secids” <subservice name> (i.e., `//blp/mktlist/secids`). The `//blp/mktlist` service is available to both BPS (Bloomberg Professional service) and NONBPS users.
8. **API Fields** — `//blp/apiflds`: Used to perform queries on the API Field Dictionary. It provides functionality similar to {FLDS <GO>}.
9. **Instruments** — `//blp/instruments`: The Instruments service is used to perform three types of operations. The first is a Security Lookup Request, the second is a Curve Lookup Request and the third is a Government Lookup Request.
10. **Page Data** — `//blp/pagedata`: Subscription-based service providing access to GPGX pages and the data they contain. The GPGX number, the monitor number, the page number and the required rows (fields) must be provided.
11. **Technical Analysis** — `//blp/tasvc`: The Technical Analysis service downloads data and brings it into an application using Bloomberg API.
12. **Curves Toolkit** — `//blp/irdctk3`: The CTK (Curves Toolkit) service allows users to interact with interest rate curves directly and retrieve the data available on the ICVS <GO> function.

5.1. SERVICE SCHEMAS

The role of the schema is to define the format of requests to the service, as well as the Events returned from that service. Within a service, one or more Event types may exist, each having its own schema. The schema is the shape of the data. For instance, market data is flat, while reference data is nested (like XML).

Each element has the following properties and attributes:

- **Name**: The name of the element.
- **Status**: ACTIVE — Available or INACTIVE — Unavailable
- **Type**: Data type of that element. This includes SEQUENCE (group), ENUMERATION, BOOL, STRING, etc.
- **Minimal Occurrence**: 0 — Optional or 1 — Required
- **Maximal Occurrence**: 1 — Element or -1 — Array

Description	Minimal Occurrence	Maximal Occurrence
Optional Field	0	1
Required Field	1	1
Array	1	-1

5.2. ACCESSING A SERVICE

All Bloomberg data provided by the Bloomberg API is accessed through a “service” (URI), which provides a schema to define the format of requests to the service and the Events returned from that service. The customer application’s interface to a Bloomberg service is a Service object.

Accessing a service is normally a two-step process:

1. Open the service using either `openService` or the `openServiceAsync` methods of the Session object.
2. Obtain a Service object for the opened service using the `getService` method of the Session object.

In both steps, the service is identified by its “name,” an ASCII string formatted as:

“//namespace/service”; (for example, “//blp/refdata” for reference data or “//blp/apiauth” for authorization functions)

Once a service has been successfully opened, it remains available for the lifetime of that Session object.

☞ *Important Note: When making a subscription-based request (via “//blp/mktdata”, “//blp/mktvwap”, or other subscription-based services), the `getService` method is not used.*

Building upon the code from the establishing a connection section, following is an example of how these service-related methods might be used.

```
<C++>
int main()
{
    SessionOptions sessionOptions;
    sessionOptions.setServerHost("10.10.10.10"); // Or specify machine name
    sessionOptions.setServerPort(8194);
    //Establish Session
    Session session(sessionOptions);
    //Attempt to Start Session
    if (!session.start()) {
        std::cerr <<"Failed to start session." << std::endl;
        return 1;
    }
    //Attempt to open reference data service
```

```

if (!session.openService("//blp/refdata"))
{
    std::cerr << "Failed to open //blp/refdata service" << std::endl;
    return 1;
}

Service authorizationService = session.getService("//blp/refdata");
}

```

In the above code, an attempt is made to open the “//blp/refdata” service and then, if successful, the user can call the `getService` method to obtain a `Service` object that will then be used when creating the request object.

5.3. MARKET DATA SERVICE

The Market Data Service (“//blp/mktdata”) enables retrieval of streaming data for securities that are priced intraday by using the API subscription paradigm. Update Messages are pushed to the subscriber once the field value changes at the source. These updates can be real time or delayed, based upon the requestors’ exchange entitlements or through setting a delayed subscription option. All fields desired must be explicitly listed in the subscription to receive their updates.

RESPONSE OVERVIEW

Once a subscription is established, the stream will supply Messages in `SUBSCRIPTION_DATA` events. The initial Message returned, known as a “summary” (initial paint) Message, will contain a value for all the available fields specified in the subscription. Subsequent Messages may contain values for some or all of the requested Bloomberg fields. A Message might contain none of the requested Bloomberg fields as the Messages are only filtered based on the fields they *could* contain rather than the fields they actually contain—many fields in the streaming events are optional. The Bloomberg API will ensure that all Messages that contain any of the fields explicitly subscribed to will be pushed to the application. Finally, the stream may return additional fields (not included in the subscription) in these Messages. These additional fields are not filtered for the purpose of speed, and their inclusion is subject to change at any time. Please note that B-PIPE users do have the option to enable field filtering, which will result in only the fields subscribed to being returned. For simplicity, this course will assume that field filtering is not applied.

The following example shows how to subscribe for streaming data.

```

<C++>
// Assume that session already exists and the "//blp/mktdata" service has
// been successfully opened.
SubscriptionList subscriptions;
subscriptions.add("IBM US Equity",
    "LAST_PRICE,BID,ASK",
    "");
subscriptions.add("/cusip/912828GM6@BGN",
    LAST_PRICE,BID,ASK,BID_YIELD,ASK_YIELD",
    "");
session.subscribe(subscriptions);

```


Some of the fields that are returned also have a null state. For example, the fields BID and ASK have values of type float and usually give positive values that can be used to populate a user's own caches. However, at times these fields will be set to a null value. For BID and ASK fields, this is usually interpreted as an instruction to clear the values in the caches. It is important to test to see if the field is null before trying to retrieve a value from it.

MARKET DATA EVENT TYPES AND SUB-TYPES

There are a number of possible market data Event types and sub-types that a subscription-based application is expected to handle. When subscribing to market data for a security, the API performs two actions:

- Retrieves and delivers a summary of the current state of the security. A summary consists of data elements known as "fields". The set of summary fields varies depending on the asset class of the requested security.
- The API streams all market data updates as they occur until subscription cancellation. About 300 market data fields are available via the API subscription interface, most of them derived from trade and quote Events.

An Event of type SUBSCRIPTION_DATA will contain a MessageType of "MarketDataEvents" that contains any of the following market data Event types (i.e., MKTDATA_EVENT_TYPE):

SUMMARY

This market data Event type Message can be any of the following market event sub-types (MKTDATA_EVENT_SUBTYPE):

- **INITPAINT** — Message is the initial paint, which comprises the most recent value for all the fields specified in the subscription as well as possibly other fields that were not included in the subscription. The inclusion of these extra fields is done to enhance performance on the Bloomberg. If the subscription is interval-based (i.e., an interval of $n > 0$), only SUMMARY INITPAINT Messages will be received every n number of seconds as the header is basically sent at the interval points with the latest tick values.
- **INTRADAY** — Message indicates a regular summary Message, usually sent near the beginning of a zero-interval-based subscription (closely after the INITPAINT SUMMARY messages). It is an update to the INITPAINT Message.
- **NEWDAY** — Sent from the Bloomberg Data Center to indicate that a new market day has occurred for the particular instrument subscribed to. It is sent after the market has closed and before the market opens the next day. Many times the first occurrence of this tick will be received an hour or two after market close. It is possible to receive more than one such tick between the market close and market open. This is the time where certain fields are re-initialized to zero, including VOLUME (the total number of securities traded in that day), to prepare for the new day.
- **INTERVAL** — Returned only when making an interval-based subscription. All messages will be of this type/sub-type. An INITPAINT Message or any QUOTE- or TRADE-type Messages will not be received.
- **DATALOSS** — Indicates that data has been lost. The library drops Events when the number of Events outstanding for delivery exceeds the specified threshold controlled by SessionOptions.maxEventQueueSize. The correlationID property attached to the DATALOSS Message identifies the affected subscription.

TRADE

This market data Event type indicates that this Event contains a trade Message and can be any of the following market data sub-types (MKTDATA_EVENT_SUBTYPE):

- **NEW** — Message contains a regular trade tick.
- **CANCEL** — Message contains a cancellation of a trade.
- **CORRECTION** — Message contains a correction to a trade.

QUOTE

This market data Event type Message can be any one of the following market Event sub-types (MKTDATA_EVENT_SUBTYPE):

- **BID** — Single BID-type field inside, along with its applicable value.
- **ASK** — Single ASK-type field inside, along with its applicable value.

- **MID** — Single MID-type field inside, along with its applicable value.
- **PAIRED** — Both single ASK- and BID-type fields inside, along with their applicable values (available only for the B-PIPE product).

☞ *For additional information, refer to the “Reference Services and Schemas Guide.”*

5.4. REFERENCE DATA SERVICE

The reference data service provides the ability to access the following Bloomberg data with the Request/Response paradigm:

- **Reference Data:** Provides the current value of a security/field pair.
- **Historical End-of-Day Data:** Provides end-of-day data over a defined period of time for a security/field pair.
- **Historical Intraday Tick Data:** Provides each tick over a defined period of time for a single security and one or more Event types.
- **Historical Intraday Bar Data:** Provides a series of intraday summaries over a defined period of time for a single security and Event type.

☞ **Note:** *Although other types of data are available under the //blp/refdata service, the aforementioned types are the **most** common and will serve as the primary focus of this module.*

5.4.1. REQUESTING REFERENCE DATA

The ReferenceDataRequest request type retrieves the current data available for a security/field pair. A list of fields is available via the Bloomberg Professional service function “**FLDS <GO>**” or by using the API fields service, which will be covered later in this module.

A ReferenceDataRequest request must specify at least one or more securities and one or more fields. The API will return data for each security/field pair or, alternatively, a Message indicating otherwise. This example shows how to construct a ReferenceDataRequest:

```
<C++>
// Assume that the //blp/refdata service has already been opened
Service refDataService = session.getService("//blp/refdata");
Request request = refDataService.createRequest("ReferenceDataRequest");
request.append("securities", "IBM US Equity");
request.append("securities", "/cusip/912828GM6@BGN");
request.append("fields", "PX_LAST");
request.append("fields", "DS002");
session.sendRequest(request, null);
```

Bulk fields and/or overrides can also be included in the request. Due to the array-like format of a bulk field, they are processed a little differently. This is covered later in the guide.

5.4.2. HANDLING REFERENCE DATA MESSAGES

A RESPONSE Message will always be returned. For large requests, one or more PARTIAL_RESPONSE Event Messages will also be returned that will include a subset of the information. A RESPONSE Message indicates the request has been fully served. This example shows how to process a Reference Data Response:

```
<C++>
void eventLoop(Session &session)
{
    bool done = false;
    while (!done) {
        Event event = session.nextEvent();
        if (event.eventType() == Event::PARTIAL_RESPONSE) {
            std::cout << "Processing Partial Response" << std::endl;
            processResponseEvent(event);
        }
        else if (event.eventType() == Event::RESPONSE) {
            std::cout << "Processing Response" << std::endl;
            processResponseEvent(event);
            done = true;
        } else {
            MessageIterator msgIter(event);
            while (msgIter.next()) {
                Message msg = msgIter.message();
                if (event.eventType() == Event::SESSION_STATUS) {
                    if (msg.messageType() == SESSION_TERMINATED ||
                        msg.messageType() == SESSION_STARTUP_FAILURE) {
                        done = true;
                    }
                }
            }
        }
    }
}

private void processReferenceDataResponse(Message msg) throws Exception {
```

```

MessageIterator msgIter(event);
while (msgIter.next()) {
    Message msg = msgIter.message();
    Element securities = msg.getElement(SEcurity_DATA);
    size_t numSecurities = securities.numValues();
    std::cout << "Processing " << (unsigned int)numSecurities
<< " securities:"<< std::endl;
    for (size_t i = 0; i < numSecurities; ++i) {
        Element security = securities.getValueAsElement(i);
        std::string ticker = security.getElementAsString(SEcurity);
        std::cout << "\nTicker: " + ticker << std::endl;
        if (security.hasElement("securityError")) {
            printErrorInfo("\tSECURITY FAILED: ",
security.getElement(SEcurity_ERROR));
            continue;
        }
        if (security.hasElement(FIELD_DATA)) {
            const Element fields = security.getElement(FIELD_DATA);
            if (fields.numElements() > 0) {
                std::cout << "FIELD\t\tVALUE"<<std::endl;
                std::cout << "-----\t\t-----"<< std::endl;
                size_t numElements = fields.numElements();
                for (size_t j = 0; j < numElements; ++j) {
                    Element field = fields.getElement(j);
                    std::cout << field.name() << "\t\t" <<
                    field.getValueAsString() << std::endl;
                }
            }
        }
        std::cout << std::endl;
    }
}
}
}

```

5.4.3. HANDLING REFERENCE DATA (BULK) MESSAGES

As discussed earlier, certain reference data fields are classified as “bulk fields.” These are indicated on “FLDS <GO>” with a “Show Bulk Data” Message where the value would normally be displayed in the right-most column. An example bulk field would be “COMPANY_ADDRESS.” This field, as is the case with all of the API bulk fields, possesses more than one piece of information (e.g., the company’s full address).

To read a bulk response, additional processing must be implemented in the EventHandler. The method below would be called once it was determined that the data response contains bulk data. This is determined by checking to see if the field element being returned is an array. Another way is to check if the DataType of that field is a SEQUENCE type. Either method can be used.

Here is what the code may look like when determining if bulk data has been received:

```
<C++>
if (security.hasElement(FIELD_DATA)) {
    const Element fields = security.getElement(FIELD_DATA);
    if (fields.numElements() > 0) {
        cout << "FIELD\t\tVALUE"<<endl;
        cout << "-----\t\t-----"<< endl;
        size_t numElements = fields.numElements();
        for (size_t j = 0; j < numElements; ++j) {
            const Element field = fields.getElement(j);
            // Checking if the field is Bulk field
            if (field.isArray()){
                processBulkField(field);
            }else{
                processRefField(field);
            }
        }
    }
}
```

Here is the code for processBulkField to read the data from the bulk response:

```
<C++>
void processBulkField(Element refBulkfield)
{
    cout << endl << refBulkfield.name() << endl ;
    // Get the total number of Bulk data points
    size_t numofBulkValues = refBulkfield.numValues();
```

```

for (size_t bvCtr = 0; bvCtr < numOfBulkValues; bvCtr++) {
    const Element  bulkElement = refBulkfield.getValueAsElement(bvCtr);
    // Get the number of sub fields for each bulk data element
    size_t numOfBulkElements = bulkElement.numElements();
    // Read each field in Bulk data
    for (size_t beCtr = 0; beCtr < numOfBulkElements; beCtr++){
        const Element  elem = bulkElement.getElement(beCtr);
        cout << elem.name() << "\t\t"
<< elem.getValueAsString() << endl;
    }
}
}

```

5.5. SOURCE REFERENCE SERVICE

The Source Reference and Tick Size subscription service (`//blp/srcref`) is used to subscribe to the source reference and tick-size data available for the specified entitlement ID. Currently, this is available per EID (FEED_EID). This allows an application to retrieve the source reference/tick-size information for all the EIDs it is entitled for. This service is available to both BPS (Bloomberg Professional service) and non-BPS users. The available source reference information includes:

- All possible values of FEED_SOURCE for the EID and a short description of the source
- Whether or not the source is a composite and all the local sources for composites
- All of the broker codes and names
- All condition codes with a short description

The syntax of the Source Reference subscription string is as follows:

```
//<service owner>/<service name>/<subservice name>/<topic>
```

where <topic> is comprised of <topic type>/<topic key>. Table below provides further details.

SOURCE REFERENCE STRING DEFINITIONS

<service owner>	For B-PIPE is "blp"
<service name>	Source Reference and Tick Size subscription service name is "/srcref"
<subservice name>	/brokercodes, /conditioncodes, /tradingstatuses or /ticksizes
<topic type>	/eid
<topic key>	EID-Number (FEED_EID1 => FEED_EID4)

Currently four subservices can be used in the subscription string.

SUBSERVICE DEFINITIONS

Subservice	Subscription String Format	Description
/brokercodes	//blp/srcref/brokercodes/eid/<eid>	List of all possible Broker codes for a specified EID
/conditioncodes	//blp/srcref/conditioncodes/eid/<eid>	List of Market Depth, Quote and Trade Condition codes for a specified EID
/tradingstatuses	//blp/srcref/tradingstatuses/eid/<eid>	List of trading statuses and trading periods for a specified EID
/ticksizes	//blp/srcref/ticksizes/eid/<eid>	List of Tick Sizes for a specified EID

Filters can be used for /conditioncodes and /tradingstatuses subscription only. Here are the possible filters available for each:

FILTERS FOR EVENTS

Filter Name (type)	Subscription String Format
Subservice Name: /conditioncodes	
TRADE	//blp/srcref/conditioncodes/eid/<eid>?type=TRADE
QUOTE	//blp/srcref/conditioncodes/eid/<eid>?type=QUOTE
MKTDEPTH	//blp/srcref/conditioncodes/eid/<eid>?type=MKTDEPTH
TRADE,QUOTE	//blp/srcref/conditioncodes/eid/<eid>?type=TRADE,QUOTE
TRADE,MKTDEPTH	//blp/srcref/conditioncodes/eid/<eid>?type=TRADE,MKTDEPTH
QUOTE,MKTDEPTH	//blp/srcref/conditioncodes/eid/<eid>?type=QUOTE,MKTDEPTH
TRADE,QUOTE,MKTDEPTH	//blp/srcref/conditioncodes/eid/<eid>?type=TRADE,QUOTE,MKTDEPTH
Subservice Name: /tradingstatuses	
PERIOD	//blp/srcref/tradingstatuses/eid/<eid>?type=PERIOD
STATUS	//blp/srcref/tradingstatuses/eid/<eid>?type=STATUS
PERIOD,STATUS	//blp/srcref/tradingstatuses/eid/<eid>?type=PERIOD,STATUS

For subscriptions without a filter, users will receive all Event types of that subservice name in the initial message as well as in subsequent daily updates. However, for subscriptions with filters, users will receive all Events in the initial message, but only specified Events in subsequent daily updates.

CODE EXAMPLE

An example can be found in the B-PIPE SDK for C++, Java and .NET. as follows:

- 1) **SourceRefSubscriptionExample** — This example demonstrates how to make a simple Source Reference subscription for one (or more) list; it sends all of the Messages to the console window. This C++ code snippet demonstrates how to subscribe for streaming source reference data.

```
const char *list = "//blp/srcref/brokercodes/eid/14005";
SubscriptionList subscriptions;
subscriptions.add(list, CorrelationId((char *)security));
session.subscribe (subscriptions);
```

☞ For additional information, refer to the “Reference Services and Schemas Guide.”

5.6. CUSTOM VWAP SERVICE

The Custom **V**olume **W**eighted **A**verage **P**rice (VWAP) Service (“//blp/mktvwap”) provides streaming VWAP values for equities. This service allows for a customized data stream with a series of overrides.

Following is a sample custom market VWAP string:

```
//blp/mktvwap/ticker/IBM US Equity?fields=VWAP&VWAP_START_TIME=10:00&VWAP_END_TIME=16:00
```

☞ *Note that it includes a single main field (VWAP) and two override field/value pairings (VWAP_START_TIME=10:00 and VWAP_END_TIME=16:00).*

User can select the single topic overload of the ADD method and pass the entire string formulated above or break down the string into topic, fields and overrides; user can then use that applicable overload of the ADD method.

The following code sample demonstrates how this can be accomplished. The response will return a Message containing a selection of VWAP fields.

```
<C++>
// Assume that session already exists and "//blp/mktvwap" service
// has been opened.
SubscriptionList subscriptions;
subscriptions.add("//blp/mktvwap/ticker/IBM US Equity",
                  "VWAP",
                  "VWAP_START_TIME=10:00&VWAP_END_TIME=16:00"
                  CorrelationId(10));
session.subscribe(subscriptions);
```

☞ *For additional information, refer to the “Reference Services and Schemas Guide.”*

5.7. MARKET DEPTH DATA SERVICE

The Enterprise Market Depth System (EMDS) is subscription-based and allows users to access a more comprehensive set of market depth data for (supported and entitled securities). It is available to both BPS (Bloomberg Professional service) and non-BPS users.

B-PIPE provides access to the entire list of “Bid” and “Ask” prices that currently exist for an instrument; this list can be known as market depth, order books or simply “Level 2” data. Most exchanges will consider this to be a separate product from their “Level 1” data (general real-time) and will charge additional fees for access to it. Thus a different EID is typically used for “Level 2.”

Generally, the “top of the book,” i.e., the price occupying the top position (position 1) of the order book, is also the “best” bid or ask. The best bid in the order book should generally be lower than the best ask, but it is possible for the ask to be higher than the bid under specific market conditions. If this occurs, then it is known as a “crossed” or “inverted” market (or book). Crossed or inverted markets can and do occur regularly under specific market conditions, most likely when the immediate matching and execution of orders has been restricted. The details of the specific conditions vary by market.

Books have three characteristics in EMDS that define them: the number of positions (rows) in the book (window size), the type of the book and the method used to update the book.

The three types of order books: Market-By-Order (MBO), Market-By-Level (MBL) and Market Maker Quote (MMQ). An exchange that operates an order book may provide only MBL data, only MBO data or both MBO and MBL data. An exchange that operates a market maker quote book will provide MMQ data. The three order/quote book update methods: Replace-By-Position (RBP), Add-Mod-Delete (AMD) and Replace-By-Broker (RBB).

☞ *For additional information, refer to the “Reference Services and Schemas Guide.”*

5.8. MARKET BAR SERVICE

The Market Bar Service (“//blp/mktbar”) provides streaming (real-time and delayed) intraday bars. This service provides the functionality to obtain intraday bars for trade volume, number of ticks, open, close, high, low and time of last trade. The service is aimed at clients wishing to retrieve HIGH/LOW prices for a specified time interval in streaming format. A subscription to a market bar requires the service to be explicitly specified in the topic. For example:

```
“//blp/mktbar/ticker/VOD LN Equity”
```

```
“//blp/mktbar/isin/GB00B16GWD56 LN”
```

The only field that can be submitted for this service is LAST_PRICE. The following code example shows a subscription to market bars for VOD LN Equity, with a bar size of 5 minutes:

```
<C++>
// Assume that the blp/mktbar service has already been opened successfully.
SubscriptionList d_subscriptions = new SubscriptionList;
d_subscriptions.add(
    “//blp/mktbar/ticker/VOD LN Equity”,
    “LAST_PRICE”,
    “bar_size=5&start_time=13:30&end_time=20:00”,
    CorrelationId(1));
d_session.subscribe(d_subscriptions);
```

The following options apply:

- **bar_size:** Length of the bar defined in minutes. The minimum supported size of the bar is 1 min. The maximum supported size of the bar is 1,440 minutes, (=24 hours).
- **start_time:** (optional): Specified in the format hh:mm. If not specified, then it is the time of session start of the security or subscription time.
- **end_time** (optional): Specified in the format hh:mm. If not specified, then it is session end time of the security.

The three types of Messages that can occur in a SUBSCRIPTION_DATA Event for this type of subscription:

- **MarketBarStart** — Occurs at a new bar. The frequency will depend upon the interval setting. A MarketBarStart will return all fields.
- **MarketBarUpdate** — Is sent on last price updates, but will include fields that have updated since the bar start or last update. Fields that are updated are VOLUME, NUMBER_OF_TICKS, TIME and CLOSE.

- **MarketBarEnd** — Occurs when the last market bar has been received (the end_time has been reached). This Message contains DATE and TIME.
- **MarketBarIntervalEnd** – Sent consistently at the end of each bar interval even if there are no TRADEs for the security at the moment.

☞ Note that no initial summary is returned for streaming intraday bars. A reference data request or a subscription is required to get an initial paint. When a market bar subscription is set to return delayed data, the market bar start Message will not be returned until the delay period has passed.

☞ For additional information, refer to the “Reference Services and Schemas Guide.”

5.9. MARKET LIST (MKTLIST) SERVICE

The Market List Service (`//blp/mktlist`) is used to perform two types of list data operations. The first is to subscribe to lists of instruments, known as “chains,” using the “chain” <subservice name> (i.e., `//blp/mktlist/chain`). The second is to request a list of all the instruments that match a given topic key using the “secids” <subservice name> (i.e., `//blp/mktlist/secids`). The `//blp/mktlist` service is available to both BPS (Bloomberg Professional service) and NONBPS users.

The syntax of the Market List subscription string is as follows:

`//<service owner>/<service name>/<subservice name>/<topic>`

where <topic> is comprised of “<topic type>/<topic key>” and <subservice name> is either “chain” or “secids.” The table below provides further details.

MARKET LIST STRING DEFINITIONS

<service owner>	For B-PIPE is “blp”	
<service name>	For subscription and one-time data is “mktlist”	
<subservice name>	/chain	Subscription-based request for a list of instruments. It can be one of a variety of types such as “Option Chains”, “Index Members”, “EID List” or “Yield Curve”. See table below for additional information and examples of each.
	/secids	Request for one-time list of instruments that match a given <topic>. It will always be “Secids List.” See table below for additional information and an example.
<topic type>	/cusip	Requests by CUSIP
	/sedol	Requests by SEDOL
	/isin	Requests by ISIN
	/bsid	Requests by Bloomberg Security Identifier
	/bsym	For requests by Bloomberg Security Symbol
	/buid	For requests by Bloomberg Unique Identifier
	/eid	For requests by Entitlement ID
	/source	For requests by Source syntax
	/bpkbl	Requests by Bloomberg parsekeyable Identifier

	/bsym	Requests by Exchange Symbol
	/ticker	Requests by Bloomberg Ticker
	/bbgid	Requests by Bloomberg Global Identifier
<topic key> ^a	The following topic types consist of source and the value of a given identifier separated by the forward slash: <source>/<identifier>	/cusip
		/sedol
		/isin
		/bpkbl
		/buid
		/bsym
	The following topic types do not require a source and consist of value alone: <Identity>	/bbgid
		/bsid
		/eid
	The following topic type consists of only a <source>	/ticker
	/source	

TABLE OF SUBSERVICE NAME EXAMPLES

Subservice Name	Example Subscription String	Topic Type**	Topic Key	Refreshes*
Option Chains	//blp/mktlist/chain/bsym/LN/BP/	/bsym	/<source>/<identifier>	No
	//blp/mktlist/chain/bsid/678605350316	/bsid	/<identifier>	No
	//blp/mktlist/chain/buid/LN/EQ0010160500001000	/buid	/<source>/<identifier>	No
	//blp/mktlist/chain/bbid/LN/EQ0010160500001000	/bbid	/<source>/<identifier>	No
	//blp/mktlist/chain/bpkbl/BP/LN Equity	/bpkbl	/<source>/<identifier>	No
	//blp/mktlist/chain/cusip/UN/594918104	/cusip	/<source>/<identifier>	No
	//blp/mktlist/chain/isin/LN/GB00B16GWD56	/isin	/<source>/<identifier>	No
	//blp/mktlist/chain/sedol/UN/2588173	/sedol	/<source>/<identifier>	No
Index List	//blp/mktlist/chain/bsym/FTUK/UKX Index; class=option	/bsym	/<source>/<identifier>	Daily
Yield Curve	//blp/mktlist/chain/bpkbl/YCMM0010 Index	/bpkbl	/<identifier>	Daily
EID List	//blp/mktlist/chain/eid/14014	/eid	/<source>	No
Source List	//blp/mktlist/chain/source/UN;secclass=Equity	/source	/<source>	No
Secids List	//blp/mktlist/secids/buid/EQ0010160500001000	/buid	/<source>/<identifier>	N/A

* Denotes whether that particular subscription (based on the <topic type> of the subscription string) will refresh and at what periodicity. For Daily refreshes, this will occur at the start of a new market day.

** The “//blp/mktlist” service currently does not support the “/ticker” topic type.

CODE EXAMPLES

Two separate examples can be found in the B-PIPE SDK for C++, Java and .NET. They are as follows:

- **MarketListSubscriptionExample**

This example demonstrates how to make a simple Market List “chain” subscription for one, or more, securities and displays all of the Messages to the console window.

- **MarketListSnapshotExample**

This example demonstrates how to make a Market List “secids” one-time request and displays the Message to the console window.

Now that the user has a better understanding of how a //blp/mktlist subscription or one-time request string is formed, the time has come to use it in the application. The following sections provide further details about how to subscribe to a chain of instruments and request a list of members.

☞ *For additional information, refer to the “Reference Services and Schemas Guide.”*

5.10. API FIELD SERVICE (APIFLDS)

The Field Information service provides details and a search capability on fields in the Bloomberg data model using the API Request/Response paradigm. Information can be retrieved in the following ways:

- **Field List Request:** Provides a full list of fields as specified by the field type (e.g., All, Static or RealTime).
- **Field Information Request:** Provides a description of the specified fields in the request.
- **Field Search Request:** Provides the ability to search the Bloomberg data model with a search string for field mnemonics.
- **Categorized Field Search Request:** Provides the ability to search the Bloomberg data model based on categories with a search string for field mnemonics.

☞ *For additional information, refer to the “Reference Services and Schemas Guide.”*

5.11. API FIELD SERVICE — FIELD LIST

A FieldListRequest request returns all of the fields defined by the field type. This loosely follows the Field Type filter option available on “FLDS <GO>” on the Bloomberg Professional service.

This example shows how to construct a FieldListRequest request.

```
<C++>
Service fieldInfoService = session.getService("//blp/apiflds");
Request request = fieldInfoService.createRequest("FieldListRequest");
request.append("fieldType", "All"); // Other options are Static and RealTime
request.set("returnFieldDocumentation", true);
std::cout << "Sending Request: " << request << std::endl;
session.sendRequest(request);
```

Possible `fieldType` values include “All” (to return all fields in the API Data Dictionary), “Static” (to return all static fields contained in the API Data Dictionary) and “RealTime” (to return all real-time fields contained in the API Data Dictionary).

A successful `FieldResponse` will contain an array of `FieldData`. The `FieldData` contains the field’s unique ID and information about the field. This example shows how to process a single `FieldResponse`. It is assumed that an Event was received with either a `RESPONSE` or `PARTIAL_RESPONSE` type prior to running this `processFieldResponse` method:

```
<C++>
MessageIterator msgIter(event);
while (msgIter.next()) {
    Message msg = msgIter.message();
    Element fields = msg.getElement("fieldData");
    int numElements = fields.numValues();
    printHeader();
    for (int i=0; i < numElements; i++) {
        printField (fields.getValueAsElement(i));
    }
    std::cout << std::endl;
}
if (event.eventType() == Event::RESPONSE) {
```

```
break;
}
```

5.12. API FIELD SERVICE — FIELD INFORMATION

A FieldInfoRequest request returns a description for the specified fields included in the request. The request requires one or more fields specified as either a mnemonic or an alpha-numeric identifier. It is also possible to specify in the request to return the documentation as per the “FLDS <GO>” function.

This example shows how to construct a FieldInfoRequest request.

```
<C++>
Service fieldInfoService = session.getService("/blp/apiflds");
Request request = fieldInfoService.createRequest("FieldInfoRequest");
request.append("id", "LAST_PRICE");
request.append("id", "pq005");
request.append("id", "ds002");
request.set("returnFieldDocumentation", true);
std::cout << "Sending Request: " << request << std::endl;
session.sendRequest(request);
```

A successful FieldResponse will contain an array of FieldData. The FieldData contains the field's unique ID and information about the field. This example shows how to process a single FieldResponse. It is assumed that an Event was received with either a RESPONSE or PARTIAL_RESPONSE type prior to running this processFieldResponse method:

```
<C++>
MessageIterator msgIter(event);
while (msgIter.next()) {
    Message msg = msgIter.message();
    Element fields = msg.getElement("fieldData");
    int numElements = fields.numValues();
    printHeader();
    for (int i=0; i < numElements; i++) {
        printField (fields.getValueAsElement(i));
    }
    std::cout << std::endl;
}
if (event.eventType() == Event::RESPONSE) {
```

```
break;
}
```

The above code snippet does not provide the code for either the `printField` or `printHeader` methods. To view these, refer to the `SimpleFieldInfoExample` example installed with the C++ API SDK.

5.13. API FIELD SERVICE — FIELD SEARCH

A `FieldSearchRequest` request returns a list of fields matching a specified search criterion. The request specifies a search string and it may also contain criteria used to filter the results. This criterion allows for the filtering by category, product type and field type.

☞ *For further information on these settings, refer to “References Services and Schemas Guide.”*

The following example shows how to construct a `FieldSearchRequest` request:

```
<C++>
Service fieldInfoService = session.getService("/blp/apiflds");
Request request = fieldInfoService.createRequest("FieldSearchRequest");
request.append("searchSpec", "last price");
Element exclude = request.getElement("exclude")
Exclude.setElement("fieldType", "Static");
std::cout << "Sending Request: " << request << std::endl;
session.sendRequest(request);

A FieldSearchRequest returns a FieldResponse just as a FieldInfoRequest does.
It is assumed that an Event was received with either a RESPONSE or
PARTIAL_RESPONSE type prior to running this processFieldResponse method:

<C++>
MessageIterator msgIter(event);
while (msgIter.next()) {
    Message msg = msgIter.message();
    Element fields = msg.getElement("fieldData");
    int numElements = fields.numValues();
    for (int i=0; i < numElements; i++) {
        printField (fields.getValueAsElement(i));
    }
    std::cout << std::endl;
}
if (event.eventType() == Event::RESPONSE) {
```

```
break;
}
```

The above code snippet does not provide the code for the `printField` method. To view this, refer to the `SimpleFieldSearchExample` example installed with the C++ API SDK.

5.14. API FIELD SERVICE — CATEGORIZED FIELD SEARCH

A `CategorizedFieldSearchRequest` request returns a list of fields matching a specified search criterion. The request specifies a search string and may also contain criteria used to filter the results. This criterion allows for the filtering by category, product type and field type.

☞ *For further information on these settings, refer to “References Services and Schemas Guide.”*

The following example shows how to construct a `CategorizedFieldSearchRequest` request:

```
<C++>
Service fieldInfoService = session.getService("//blp/apiflds");
Request request =
fieldInfoService.createRequest("CategorizedFieldSearchRequest");
request.append("searchSpec", "last price");
Element exclude = request.getElement("exclude")
Exclude.setElement("fieldType", "Static");
Request.set("returnFieldDocumentation", false);
std::cout << "Sending Request: " << request << std::endl;
session.sendRequest(request);
```

A successful `CategorizedFieldResponse` will contain an array of `CategoryData` that contains a flattened representation of the matching fields arranged by the category tree. This example shows how to process a single `CategorizedFieldResponse`.

```
<C++>
MessageIterator msgIter(event);
while (msgIter.next()) {
    Message msg = msgIter.message();
    if (msg.hasElement(FIELD_SEARCH_ERROR)) {
        msg.print(std::cout);
        continue;
    }
    Element categories = msg.getElement("category");
    int numCategories = categories.numValues();
```



```

for (int catIdx=0; catIdx < numCategories; ++catIdx) {
    Element category = categories.getValueAsElement(catIdx);
    std::string Name = category.getElementAsString("categoryName");
    std::string Id = category.getElementAsString("categoryId");
    std::cout << "\n Category Name:" << padString (Name, CAT_NAME_LEN) <<
        "\tId:" << Id << std::endl;
    Element fields = category.getElement("fieldData");
    int numElements = fields.numValues();
    printHeader();
    for (int i=0; i < numElements; i++) {
        printField (fields.getValueAsElement(i));
    }
}
std::cout << std::endl;
}
if (event.eventType() == Event::RESPONSE) {
    break;
}

```

5.15. INSTRUMENTS

//blp/instruments: The Instruments service is used to perform three types of operations. The first is a Security Lookup Request, the second is a Curve Lookup Request and the third is a Government Lookup Request. Instruments from a common source (e.g., NASDAQ) will share an EID. For example, MSFT UQ Equity and INTC UQ Equity both come from NASDAQ and have EID 14005 (if requested by someone with Level 1 access).

☞ *For additional information, refer to the “Reference Services and Schemas Guide.”*

5.16. PAGE DATA

The Page Data service of the API provides access to GPGX pages and the data they contain. This is a subscription service that requires that the GPGX number, the monitor number, the page number and the required rows (fields) be provided.

The topic is constructed as follows:-

0708/012/0001

where:

0708 is the **GPGX number**

012 is the **monitor number**

0001 is the **page number**

An array of strings is used to specify the rows on the page that are of interest. These can be specified as individual rows, multiple rows separated by commas or ranges of rows, as follows:

String	Rows Specified
"1"	The first row on the page
"1,2,3"	Rows 1,2 and 3 on the page
"1,6-10,15,16"	Row 1, rows 6 to 10 and rows 15 and 16

The following example shows how to create a subscription and demonstrates how the subscription fields are used to pass the rows the user wants to subscribe to.

```
std::string topic = "0708/012/0001";
std::string page = "blp/pagedata/" + topic;

subscriptions.add(page.c_str(),
    "15-18", // subscribe to rows 15 to 18
    new CorrelationID(topic));
```

RESPONSE BEHAVIOR

Once a subscription has been created and the subscription status Messages have been processed, two Event types might be received:

5.17. TECHNICAL ANALYSIS

Technical Analysis is a method of evaluating securities by analyzing statistics generated by market activity, such as past prices and volumes. Technical analysts do not attempt to measure a security's intrinsic value, instead, they use charts and other tools to identify patterns that can suggest future activity. The Technical Analysis Service enables users to download this data and bring it into their application using Bloomberg API.

The table below displays details on the different Technical Analysis data types:

Data Type	Description
Historical End-of-Day	End-of-day data for a specified period of time in increments of days, weeks, months, quarters or years.
Intraday	Intraday data for a specified period of time in increments of minutes. Based on Bid, Ask or Trade Events, data such as open, high, low, close and volume can be retrieved for the interval of time specified.
Real-time	Real-time data and Events.

5.18. HISTORICAL END-OF-DAY STUDY REQUEST

The Historical study request enables the retrieval of end-of-day technical analysis data for a specified security and study attributes over the specified time periods of daily, weekly, monthly, biannually and annually. Each Historical study request can submit only a single instrument.

```
Service tasvcService = session.getService("//blp/tasvc");
Request request = tasvcService.createRequest("studyRequest");

// set security name
request.getElement("priceSource").
    getElement("securityName").setValue("IBM US Equity");

// set historical price data
request.getElement("priceSource").
    getElement("dataRange").setChoice("historical");
Element historicalEle = request.getElement("priceSource").
    getElement("dataRange").getElement("historical");

// set study start date
historicalEle.getElement("startDate").setValue("20100501");

// set study end date
historicalEle.getElement("endDate").setValue("20100528");

// DMI study example - set study attributes
request.getElement("studyAttributes").setChoice("dmiStudyAttributes");
Element dmiStudyEle = request.getElement("studyAttributes").
    getElement("dmiStudyAttributes");

// DMI study interval
dmiStudyEle.getElement("period").setValue(15);

// set historical data price sources for study
dmiStudyEle.getElement("priceSourceLow").setValue("PX_LOW");
dmiStudyEle.getElement("priceSourceClose").setValue("PX_LAST");
```

RESPONSE BEHAVIOR

A successful `studyResponse` holds information on the requested security. It contains a `studyDataTable` with one `studyDataRow` for each interval returned.

```

void processResponseEvent(Message msg)
{
    Element security = msg.getElement(SEcurity_NAME);
    std::string ticker = security.getValueAsString();
    std::cout << std::endl;
    std::cout << "Ticker: " << ticker << std::endl;
    if (security.hasElement("securityError"))
    {
        printErrorInfo("\tSECURITY FAILED: ", security);
        return;
    }
    Element fields = msg.getElement(STUDY_DATA);
    int numValues = fields.numValues();
    if (numValues > 0)
    {
        for (int j = 0; j < numValues; ++j)
        {
            Element field = fields.getValueAsElement(j);
            int numElems= field.numElements();
            for (int k =0; k < numElems; k++)
            {
                Element element = field.getElement(k);
                std::cout << "\t" << element.name() + " = "
                    + element.getValueAsString() << std::endl;
            }
            std::cout << std::endl;
        }
    }
}

```

```

    }
}

```

5.19. INTRADAY BAR STUDY REQUEST

The Intraday Bar-type study request enables the retrieval of summary intervals of intraday technical analysis data for a specified study attributes for five Event types, TRADE, BID, ASK, BEST_BID and BEST_ASK, over a period of time. Each Intraday study request can submit only a single instrument. In addition, the Event type, interval and date/time start and end points in UTC must be specified.

```

Service tasvcService = session.getService("//blp/tasvc");
Request request = tasvcService.createRequest("studyRequest");

// set security name
request.getElement("priceSource").
    getElement("securityName").setValue("IBM US Equity");

// set to intraday
Element intradayEle = request.getElement("priceSource").
    getElement("dataRange").getElement("intraday");
// set intraday price data

// intraday event type
intradayEle.getElement("eventType").setValue("TRADE");

intradayEle.getElement("interval").setValue(60); // intraday interval

// set study start date
intradayEle.getElement("startDate").setValue("2010-05-26T13:30:00");

// set study end date
intradayEle.getElement("endDate").setValue("2010-05-27T13:30:00");

// smavg study example - set study attributes
request.getElement("studyAttributes").setChoice("smavgStudyAttributes");
Element smavgStudyEle = request.getElement("studyAttributes").

```

```
getElement("smavgStudyAttributes");
smavgStudyEle.getElement("period").setValue(15); // SMAVG study interval
smavgStudyEle.getElement("priceSourceClose").setValue("close");
```

RESPONSE BEHAVIOR

A successful studyResponse holds information on the requested security. It contains a studyDataTable with one studyDataRow for each bar interval returned.

```
void processResponseEvent(Message msg)
{
    Element security = msg.GetElement(SEcurity_NAME);
    string ticker = security.GetValueAsString();
    std::cout << std::endl;
    std::cout << "Ticker: " << ticker << std::endl;
    if (security.hasElement("securityError"))
    {
        printErrorInfo("\tSECURITY FAILED: ", security);
        return;
    }
    Element fields = msg.getElement(STUDY_DATA);
    int numValues = fields.numValues();

    if (numValues > 0)
    {
        for (int j = 0; j < numValues; ++j)
        {
            Element field = fields.GetValueAsElement(j);
            int numElems= field.numElements();
            for (int k = 0; k < numElems; k++)
            {
                Element element = field.getElement(k);
                Std::cout << std::endl;
            }
        }
    }
}
```

```

        std::cout <<"\t" << element.name() << " = "
        << element.GetValueAsString() << std::endl;
    }
}
}
}
}

```

5.20. REAL-TIME STUDY REQUEST

The Real-time study request provides the ability to subscribe to real-time technical analysis data points for specified study field attributes and period. Each real-time study subscription can subscribe only to a single study field.

Assume that session already exists and the “//blp/tasvc” service has been successfully opened.

```

SubscriptionList subscriptions = new SubscriptionList;
subscriptions.Add("//blp/tasvc/ticker/IBM US Equity?fields=WLPR" \
    "priceSourceClose=LAST_PRICE&" \
    "priceSourceHigh=HIGH&" \
    "priceSourceLow=LOW&" \
    "periodicitySelection=DAILY&" \
    "period=14"
    ,new CorrelationID("IBM US Equity_WLPR"));
session.subscribe (subscriptions);

```

RESPONSE BEHAVIOR

Once a subscription is established, the stream will supply Messages in SUBSCRIPTION_DATA Events. In addition to the study field subscribed, users may receive additional study fields, which were not subscribed, in these Messages. These additional fields are not filtered for the purpose of speed and their inclusion is subject to change at any time.

6. Best Practices

This includes information on how data usage limits work so as developers build apps they can keep this information in mind to avoid hitting data limits. This section talks about stability, where testing is the key to stable code, as is checking return values. Following are some recommended best practices to follow when developing Bloomberg API-based applications:

6.1. NAME CLASS

- This is an efficiency class for strings and is similar to a guaranteed interned string. Use the Name class to define strings used in the application.
- “Name” is very efficient if not overused. Remember that once constructed, no way is available to free the allocated internal memory.
- The Bloomberg API is optimized to handle Names. For example, the GetElement and GetValue methods use hash table lookups to quickly retrieve the data from the Message structure. If passing a String instead of a Name, converting to a hash value is required. Therefore, if pre-computing the hash value by defining as a Name on program start-up, it will first check if there already is an instance. If so, it will return it. Otherwise, it creates a new one and returns it.
- Store all element name strings as: static const Name.
- Once the schema is downloaded, names are created for each Enum value. Can be used to create static names for each value and allows efficient name comparison, instead of string comparison.

6.2. SESSION OBJECT

- Sessions are essentially Data Stream connections; note that these are logical connections and the API supports failover between physical connections. During failover, API will handle re-subscriptions.
- It is recommended to use two Sessions to avoid delaying a thin fast stream with a fat slow one. Have one Session object for real-time data and one for reference data requests, with a possible additional Session for large historical requests.
- Opening and closing a Session is expensive for both the client’s application and for Bloomberg servers.
- Check admin Messages and handle all cases accordingly, for example: calling session.Stop() is not a good idea on a SESSION_TERMINATED Message, the session is already stopped

6.3. EVENTQUEUE

- EventQueues can replace the default Session Event queue per Request. The result of a Request can occur in several Partial Events.
- Asynchronous requests can be made blocking a Request after it is made passing an EventQueue, all Events for that Request will appear on that EventQueue or once the Request is complete.
- The function can return the results from parsing all the Events.
- An Event is a container for Messages and owns all memory for Messages.
- As it can hold many Messages, it can be quite large.

👉 **Note:** In C++, Messages point inside an Event and the lifespan of a Message must not exceed its Event’s lifespan.

EXAMPLE OF USING AN EVENTQUEUE

```

C++
Send Request using EventQueue:
    Request request = refDataService.createRequest("ReferenceDataRequest");
    ...
    ...
    EventQueue eventQueue;
    CorrelationId cid(this);
    session.sendRequest(request, cid, &eventQueue);
    while (true)
    {
        Event eventObj = eventQueue.nextEvent();
        MessageIterator msgIter(eventObj);
        while (msgIter.next()) {
            Message msg = msgIter.message();
            // Process Message
            ...
            ...
        }
        if (eventObj.eventType() == Event::RESPONSE)
        {
            break;
        }
    }
}

```

6.4. GETTING VALUES FROM ELEMENTS

The `getAsString` method converts a native type to a string. If a native object is required, use the native type–getter method—it is a much more efficient than getting the value as a string and then converting the string back to a native type.

In C++, the string returned should be copied. It is a `const char *` and a pointer into its `Event`. The following is more efficient as it doesn't cause an exception like `getElementAsXXXX`:

```

Element E;

int err=getElement(&E, Name);

if (!err)int valerr=E.getValueAs(&MYVAL);

```

(Name) obviates the need for a second lookup through a call to the `hasElement()` method, whose Boolean argument determines if Null is treated the same way as the field not existing at all.

6.5. MAXEVENTQUEUE SIZE

The default for `SessionsOptions.maxEventQueueSize` is 10,000 and this should always be higher than the number of subscriptions. If more Events exist, they will be dropped. The notification is: `SlowConsumer(Hi/Lo)Watermark`. Higher values reduce the chance of clearing the eventQueue and increase potential unmonitored latency and there is a risk of consuming very high amounts of memory.

6.6. MESSAGE ITERATOR

It is recommended not to exit an Event processing loop with a return or break but to continue to the next iteration. Events can have multiple Messages and breaking the loop, or returning from the callback without checking for more Messages, will prevent handling of later Messages in the Event and they will be lost. Breaking the loop will appear as if ticks are being dropped.

6.7. CORRELATIONID OBJECT

This is used to match a particular request with its returned data (request or subscription). It can be assigned any object. Rather than using a string as the CorrelationID and then using that string as a lookup key in a container of Data objects, use the Data object itself. It may still be necessary to put the Data object in a container, but callbacks can directly access the object.

REQUESTS

These can be assigned to any object. Bundle requests whenever possible, as multiple securities in one single request will be processed more efficiently than a single security per request.

LOGGING

It is a best practice to log all non-data Messages, even in production. Data Messages are SUBSCRIPTION_DATA, RESPONSE and FINAL_RESPONSE. Under normal situations, there should be relatively few non-data Events. When an error occurs, these Messages contain important debugging information. The entire Message should be logged. It might be helpful during development to log data Messages, but this generally should not be done in a production environment.

DATA LIMITS

Usage is metered by unique subscriptions and reference data hits; it is best to factor this into application design. Unique subscriptions are the number of subscriptions that have the same security, with possible changes in fields or options. Twenty requests for “IBM US Equity” would count as one even if the fields are different or some are interval or delayed.

Reference data hits refer to the number of securities, multiplied by the number of fields requested, and are counted for each request. The same exact request for 3 fields across 7 securities made 5 times would count as $3 * 7 * 5$, or 105 hits. The number of rows returned is not relevant, so tick history for 2 fields on IBM would count as 2, even if thousands of rows are returned. It is best to centralize and not repeat requests that are unlikely to change. Requesting the latest frequently changing fields like BID and ASK every minute should be replaced by a subscription for those fields with an interval of 1 minute—this will dramatically reduce the number of reference data hits.

6.8. DATETIME

DateTime is basically a struct, and in cases where it is convertible to a native datetime type it is generally better to convert it to a native type. However, in some cases, it is not convertible, for example, it might only hold a month. Store it in a standard string format if it does not convert and if it is only displayed and never used for calculations. The format for String conversion to DateTime is based on field type and must be exact:

Type	API Name	Format	Example
DateTime	BLPAPI_DATATYPE_DATETIME	yyyy-mm-ddThh:mm:ss	2011-09-29T14:59:59
Date	BLPAPI_DATATYPE_DATE	yyyy-mm-dd	2011-09-29
Time	BLPAPI_DATATYPE_TIME	hh:mm:ss.xxx	14:59:59.123

For DateTime and Time: Optionally .xxx can be added for milliseconds to— see Time above and +tt:tt can be added for time zone offset (no spaces)

6.9. IF CHAINS

It is common in example code to see:

```
if (msg.hasElement(ELEMENT))
{
    myObject.ELEMENT=msg.getAsString(ELEMENT);
}
```

and then repeat that for every element

In practice, it is more efficient in many places to iterate through all fields and check if the Message has the required field:

```
foreach element e in ms
{
    if (e.name() == ELEMENT)
    {
        myObject.ELEMENT=e.getValueAsString();
    }
}
```

This needs to be a per-case evaluation. Based on the number of fields required vs. the number of fields expected in the Event,

Where `ELEMENT`

Can be any Bloomberg or Local Schema Field string stored in a Name.

6.10. INVARIANTS: UN-OPTIMIZED

It is especially the case when dealing with reference data that a certain sequence of calls must be used for getting data; it will look like this:

```

MessageIterator iter(event);
while(iter.next())
{
    Message msg=iter.message();
    for (int idx = 0;
        idx < msg.asElement().getElement("securityData").numValues();
        idx++)
    {
        if (msg.asElement().getElement("securityData")
            .getValueAsElement(idx)
            .getElement("fieldData")
            .hasElement("ELEMENT"))
        {
            myObject.ELEMENT=
                msg.asElement().getElement("securityData")
                    .getValueAsElement(idx)
                    .getElement("fieldData")
                    .getElementAsString("ELEMENT");
        }

        if (msg.asElement().getElement("securityData")
            .getValueAsElement(idx)
            .getElement("fieldData")
            .hasElement("ELEMENT2"))
        {
            myObject.ELEMENT2=
                msg.asElement().getElement("securityData")
                    .getValueAsElement(idx)
                    .getElement("fieldData")
                    .getElementAsString("ELEMENT2");
        }
        ...
    }
}

```

6.10.1. INVARIANTS: CORRECTED

```

C++
// This should be moved up to outer loops as follows:
Element securities;           // All Security Data
Element security;           // Data for just one Security
Element data;               // All field data for that security
Element field;              // Data for one field
String Element;            // Temp storage for String data

    MessageIterator iter(event);
    while(iter.next()) {
        Message msg=iter.message();
        if (msg.GetElement(&securities, SECURITY DATA) != 0) // non 0
means error

    next;
        int maxIndex=securities.numValues();
        for (int idx = 0; idx < maxIndex; ++idx)

        {

            security=securities.getValueAsElement(idx);
            if (security.GetElement(&data, FIELD DATA) != 0) // non 0 means
error
            next;
            if (data.GetElement(&field, ELEMENT))           // Note this is a Name now
                if (field.getValueAs(&Element) == 0) // 0 means no error

                Object.ELEMENT=Element;

            if (data.GetElement(&field, ELEMENT2))           // Note this is a Name now
                if (field.getValueAs(&Element) == 0) // 0 means no error

                Object.ELEMENT2=Element;

            ...
        }
    }
- Where ELEMENT, ELEMENT2
Can be any Bloomberg or Local Schema Field string stored in a Name}

```

7. Event Handling

Asynchronous and synchronous Event handler code is quite similar. However, there are a few notable differences between the two:

SYNCHRONOUS

- Simpler of the two modes to code.
- Blocks until an Event becomes available.
- Uses the `Session.NextEvent` method to obtain the next available Event object.

ASYNCHRONOUS

- Recommended mode.
- Does not block until an Event becomes available. Instead, the Event handler is triggered by a callback.
- Requires that an `EventHandler` object be supplied when creating a `Session`.

The `Session` class also provides ways of handling Events synchronously and asynchronously. The simpler of the two is synchronous Event handling, which requires calling `Session.NextEvent` to obtain the next available Event object. This method will block until an Event becomes available. A synchronous request model is well-suited for single-threaded customer applications.

The alternative and preferred method to handling data request Events is asynchronously, which is accomplished by supplying an `EventHandler` object when creating a `Session`. In this case, the user-defined `processEvent` method in the supplied `EventHandler` will be called by the Bloomberg API when an Event is available. For instance, the C++ API signature for `processEvent` method is:

```
public void processEvent(Event event, Session session) // No exceptions are thrown
```

The callbacks to the `processEvent` method will be executed by a thread owned by the Bloomberg API, thereby making the customer application multi-threaded; consequently, customer applications must ensure that data structures and code accessed from both its main thread and from the thread running the `EventHandler` object are thread-safe.

The mode of a `Session` is determined when it is constructed and cannot be changed subsequently.

As previously discussed, there are two modes for handling data requests: asynchronously and synchronously. These choices are mutually exclusive:

- If a `Session` is provided with an `EventHandler` when it is created (i.e., asynchronous mode), then calling its `NextEvent` method (used for synchronous mode) will throw an exception.
- If no `EventHandler` is provided with a `Session` (i.e., synchronous mode), then the only way to retrieve an Event object is by calling its `NextEvent` method.

7.1. ASYNCHRONOUS VS. SYNCHRONOUS

Let's review two scenarios to understand the difference between asynchronous and synchronous:

1. B requires the telephone number of C and needs to obtain it from A.
2. B has the option of waiting for C if either C is absolutely essential for B making any further work progress or because this is the way B operates in general. Alternatively, B can instruct A to indicate receipt of the C information, while, in the meantime, B continues with something else.

The first situation illustrates synchronicity where a requester (B) of information will wait (block) until that information is available. Programmatically speaking this may be because the data requested is essential to complete the next operation or because the program is procedural in nature and follows the same fixed serialized pattern as Boss B's work mode.

The second situation is an example of asynchronicity. In this case, the requester (B) will not wait (block) after the initial request is made, but instead will continue to the next program statement after data is requested.

It is up to the developer to request data synchronously or asynchronously. Here are two possible scenarios for each:

Synchronous

- Application needs to make some authentication-type request before allowing data requests.
- Data from a basket of securities must be applied to a custom algorithm before storing the results in a local database.

Asynchronous

- Application program is Event-driven.
- Application is a monitoring application requiring real-time pricing data as a source.

The Bloomberg API is fundamentally asynchronous. Applications initiate operations and subsequently receive Event objects to notify them of the results. However, for developer convenience, the Session class also provides synchronous versions of some operations. For instance, the Start and OpenService methods encapsulate the waiting for the Events and make the operations appear synchronous. The equivalent asynchronous methods for these methods are StartAsync and OpenServiceAsync, respectively.

7.2. MULTIPLE SESSIONS

Many applications will use only a single Session (connection). However, the Bloomberg API allows the creation of multiple Session objects. Multiple instances of the Session class contend for nothing and thus allow for efficient multi-threading.

For example, an application can increase its robustness by using multiple Session objects to connect to different communication-process instances.

Another example would be an application that may require both large heavyweight Messages that need much processing as well as small Messages that can be quickly processed. If both were obtained through the same Session, the processing of the large messages would increase latency on the lightweight Messages. This can be mitigated by handling the two categories of data with different Session objects and different threads.

8. Message Types

Once a “//blp/mktdata” subscription is established, the stream will supply Messages in SUBSCRIPTION_DATA Events. Each Message will contain a MKTDATA_EVENT_TYPE and MKTDATA_EVENT_SUBTYPE value. For instance, the initial Message returned, identified with SUMMARY/INITPAINT values, will contain a value for all the fields specified in the subscription. Subsequent Messages may contain values for some or all of the requested Bloomberg fields. It is possible for a Message to contain none of the requested Bloomberg fields as the Messages are only filtered based on the fields they *could* contain rather than the fields they actually contain—many fields in the streaming events are optional. The Bloomberg API will ensure that all Messages that contain any of the fields users have explicitly subscribed for are pushed to their application. Finally, the stream may return additional fields in these Messages that were not included in the subscription. These additional fields are not filtered for the purpose of speed, and their inclusion is subject to change at any time.

Here is a list of the common MKTDATA_EVENT_TYPE/MKTDATA_EVENT_SUBTYPE value pairings, along with descriptions that users may see in the SUBSCRIPTION_DATA Event Messages:

Market Event Pairings	Description
MKTDATA_EVENT_TYPE = SUMMARY MKTDATA_EVENT_SUBTYPE = NEWDAY	New Day Turnaround: Includes any subscribed fields that should be blanked out prior to new day open along with a refresh of static values needed to maintain statistics and process other Messages.
MKTDATA_EVENT_TYPE = SUMMARY MKTDATA_EVENT_SUBTYPE = INITPAINT	Initial Paint: Contains an initial value for all the fields specified in the subscription.f
MKTDATA_EVENT_TYPE = SUMMARY MKTDATA_EVENT_SUBTYPE = INTRADAY	Intraday: Contains an update to the initial paint (INITPAINT).
MKTDATA_EVENT_TYPE = SUMMARY MKTDATA_EVENT_SUBTYPE = INTERVAL	Interval: Returned only when making an interval-based subscription. All Messages will be of this type/sub-type. Users will not receive an INITPAINT Message.
MKTDATA_EVENT_TYPE = SUMMARY MKTDATA_EVENT_SUBTYPE = DATALOSS	Data Loss: See What Is the Difference Between a DataLoss Admin Message and a SUMMARY/DATALOSS Message?. for further information
MKTDATA_EVENT_TYPE = TRADE MKTDATA_EVENT_SUBTYPE = NEW	New Trade: Contains a regular trade tick.
MKTDATA_EVENT_TYPE = TRADE MKTDATA_EVENT_SUBTYPE = CANCEL	Cancel Trade: Contains a cancellation of a trade.
MKTDATA_EVENT_TYPE = TRADE MKTDATA_EVENT_SUBTYPE = CORRECTION	Correct Trade: Contains a correction to a trade.
MKTDATA_EVENT_TYPE = QUOTE MKTDATA_EVENT_SUBTYPE = BID	Bid Quote: Should contain a single BID-type field.
MKTDATA_EVENT_TYPE = QUOTE MKTDATA_EVENT_SUBTYPE = ASK	Ask Quote: Should contain a single ASK-type field.
MKTDATA_EVENT_TYPE = QUOTE MKTDATA_EVENT_SUBTYPE =MID	Mid Quote: Should contain a single MID-type field.
MKTDATA_EVENT_TYPE = QUOTE MKTDATA_EVENT_SUBTYPE = PAIRED	Paired Quote: Should contain both a BID- and ASK-type fields.
MKTDATA_EVENT_TYPE = MARKETDEPTH MKTDATA_EVENT_SUBTYPE = TABLE	MarketDepth: Should contain values for market depth fields from levels 1 through 5 (for Desktop and Server API) and levels 1 through 10 (for B-PIPE) if the user included one or more of the market depth fields in the subscription. For a list of these fields, expand item 16 in the first section of the Field, Security & General Data Topics page.

9. Error Codes

9.1. COMMON EXCEL ERROR MESSAGE CODES

- **#N/A Authorization:** Indicates missing authorization while trying to access a service for which not privileged for, if no user information in request (valid user handle is required in every request/subscribe call) or Platform not enabled in EMRS for particular user.
- **#N/A Connection:** Connection between the BBCOMM (which handles the actual transfer of data from our database to the user's PC) and the RTD (which passes requests from Excel to the BBCOMM) has been severed.
- **#N/A Published Data:** N Succeeded, N Failed: Publish Event succeeded with a count of how many specific fields succeeded and how many failed. At least one field must have "succeed."
- **#N/A Requesting Data...:** Displays while waiting for formula result to be returned.
- **#N/A Field Not Applicable:** Field is not applicable for the security. For example, trying to retrieve the yield on a stock or the dividend per share on a bond.
- **#N/A Invalid Security:** Security is not recognized.

9.2. MESSAGE RETURN CODES (V3 API ONLY)

9.2.1. GENERAL

Event Name	Message Type	Category	Scenario
ADMIN	SlowConsumerWarning		Indicates client is slow. NO category/subcategory.
ADMIN	SlowConsumerWarningCleared		Indicates client is not slow anymore. NO category/subcategory.
ADMIN	DataLoss		Generated when the event queue overflows and events must consequently be dropped. Messages of this type will be generated after a `SlowConsumerWarning` and before any subsequent `SlowConsumerWarningCleared`, but unlike those messages, events containing `DataLoss` will be appended at the point in the queue where events are dropped (and might thus be pulled from the queue after a `SlowConsumerWarningCleared` which was generated at a later point in time). A single `DataLoss` message may represent a large of number of lost events; see the type description for information on the meta-data captured to describe the missing events.
ADMIN	RequestTemplateAvailable		The request template has been fully prepared, and will be subject to optimized processing when sent, if fully processed before the next `RequestTemplatePending` message. For subscription service snapshot requests, optimized processing entails delivering topic recaps directly from a hot cache.
ADMIN	RequestTemplatePending		The request template is in the process of being prepared, and may not receive optimized processing if sent. This message may be generated either as a result of a new template being prepared for the first time, or as a result of a change to processing infrastructure that requires

			re-preparation (e.g. the cache associated with a snapshot template must be migrated from one machine to another due to failover). When the preparation is complete, a subsequent 'RequestTemplateAvailable' message will be delivered.
ADMIN	RequestTemplateTerminated		The request template is no longer valid; any subsequent attempt to send the request will result in failure.
SESSION_STATUS	SessionStarted		Session has been started successfully.
SESSION_STATUS	SessionTerminated	IO_ERROR	Session has been terminated.
SESSION_STATUS	SessionStartupFailure	IO_ERROR	Session has failed to start.
SESSION_STATUS	SessionConnectionUp	IO_ERROR	Session is up either because Session.Start() was called or the connection between the application and the Bloomberg Communication Server process (e.g., ServerApi, B-PIPE) has been reestablished.
SESSION_STATUS	SessionConnectionDown	IO_ERROR	Session is down either because Session.Stop() was called or the connection between the application and the Bloomberg Communication Server process (e.g. ServerApi, B-PIPE) has been lost.
SESSION_STATUS	ServiceOpened		Service has been opened successfully.
SESSION_STATUS	ServiceOpenFailure	IO_ERROR	Service has failed to open (I/O Error).
SESSION_STATUS	ServiceOpenFailure	UNCLASSIFIED	Service has failed to open (Other).
SESSION_STATUS	SessionClusterInfo		Delivered when the SDK is connected to a cluster and receives initial cluster configuration information from the Bloomberg infrastructure. The message includes the name of the cluster and a list of end points within.
SESSION_STATUS	SessionClusterUpdate		This is delivered when the SDK is connected to a cluster and receives updated configuration information about the cluster from the Bloomberg infrastructure. The message includes the name of the cluster and lists of end points that have been added/removed.

9.2.2. **API ERROR CODE NUMBER**

The table below lists the API Error code with the number:

Error Code	Number
SUCCESS	0
INTERNAL_ERROR	100
INVALID_USER	101
NOT_LOGGED_ON	102
INVALID_DISPLAY	103
ENTITLEMENT_REFRESH	105
INVALID_AUTHTOKEN	106 (ONLY USED BY SAUT)
EXPIRED_AUTHTOKEN	107
TOKEN_IN_USE	108

9.2.3. **//BLP/APIAUTH**

(AUTHORIZATION_STATUS, REQUEST_STATUS, RESPONSE and PARTIAL_RESPONSE Events) [Server API and B-PIPE]

Request	Message Type	Category	Sub-Category	Scenario
Authorization Request	AuthorizationSuccess			User was authorized successfully.
	AuthorizationFailure	NO_AUTH	NOT_LOGGED_IN	User is not logged in to Bloomberg.
	AuthorizationFailure	BAD_ARGS	INVALID_USER	Invalid User ID
	ResponseError	NO_AUTH	CROSS_FIRM_AUTH	Valid User ID belonging to different firm
	AuthorizationFailure	NO_AUTH	INVALID_DISPLAY	Invalid Display (when IP is specified)
	AuthorizationFailure	NO_AUTH	TOKEN_EXPIRED	Timeout waiting for input or expired token
	AuthorizationFailure	NO_AUTH	BAD_AUTH_TOKEN	Bad unparsable token supplied
	AuthorizationFailure	NO_AUTH	CANCEL_BY_USER	User cancels request (Launchpad).
	AuthorizationFailure	NO_AUTH	ENTITLEMENTS_MISMATCH	UserAsidEquivalence check failed.
	ResponseError	BAD_ARGS	N/A	No token and IP specified.
	EntitlementChanged	N/A	N/A	User has logged off and then back on to the Bloomberg Professional service. The user's Identity object remains valid.
	EntitlementChanged	N/A	N/A	Entitlements of the User/Application have been changed in EMRS. Usually needs hour to take effect and, therefore, to generate Message. The user/application's Identity object remains valid. Message = "Administrative Action"
	AuthorizationRevoked	NO_AUTH	INVALID_DISPLAY	A user logs in to a Bloomberg Professional service other than the one on the PC running application.
AuthorizationFailure	NOT_AVAILABLE	NOT_AVAILABLE_API	When user uses an API that is either deprecated or passes parameters in an authorization request that are not supported for the specific product in question. For example, emrsname + IP authorization is not supported for ServerApi. Similarly UUID+IP authorizations are not supported on platforms. A descriptive error	

Request	Message Type	Category	Sub-Category	Scenario
				Message is returned in the latter case.
	AuthorizationRevoked	NO_AUTH	LOCKOUT	User locked out of the Bloomberg Professional service. [Read “How Do I Handle an AuthorizationRevoked Message?”]

Message Type	Category	Sub-Category	Scenario
Authorization Update	NO-AUTH	INVALID_DISPLAY	User logged in to another Bloomberg Professional service.
Authorization Update	NO-AUTH	LOCKOUT	User locked out of Bloomberg Professional service. Click here for further details.
Authorization Update	UNCLASSIFIED	CANCELLED_BY_SERVER	Authorization cancelled by the server through EMRS administrator.
Authorization Request	NO-AUTH	NO_APP_PERM	User not permitted to use the application.
Authorization Request	NO-AUTH	INVALID_ASID_TYPE	Requested authorization type not supported for this ASID type.
Authorization Request	NO-AUTH	CREDENTIAL_REUSE	User’s authorization token has been used by another instance.
Authorization Request	NO-AUTH	EXPIRED_AUTHTOKEN	The token has expired. User must regenerate the token and authorize.
Authorization Request	LIMIT	MAX_DEVICES_EXCEEDED	The maximum number of devices for this seat type has been exceeded.
Authorization Failure	LIMIT	MAX_AUTHORIZATIONS_EXCEEDED	Exceeded maximum number of simultaneous authorizations.
Authorization Update	NO-AUTH	EMRS_ENTITY_ASID_MISMATCH	Entity/ASID delivery point not enabled in EMRS. User gets this if a failure is dynamically detected because someone changed EMRS and an existing authorization is affected after the authorization had been successfully made.
Authorization Failure	NO-AUTH	EMRS_ENTITY_ASID_MISMATCH	Entity/ASID combination not enabled in EMRS. User gets this if failure detected at authorization time.
Authorization Failure	NO-AUTH	EMRS_IPRANGE_MISMATCH	Application IP mismatch with EMRS IP ranges.
Authorization Failure	NO-AUTH	EMRS_DATAFEED_DISABLED	User or Application not enabled for datafeed (B-PIPE) access in EMRS and attempting to authorize using a B-PIPE.
Authorization Failure	NO-AUTH	EMRS_PLATFORM_DISABLED	User or Application not enabled for platform access in EMRS and attempting to authorize using a DDM.
Authorization Failure	NO-AUTH	INVALID_DELIVERY_POINT	Application has no instance created for the B-PIPE instance (delivery point) in EMRS.
Authorization Failure	NO-AUTH	IP_NOT_IN_RANGE	Application is authorizing from a machine whose IP is being prevented by the IP Restrictions configured in EMRS.
Authorization Revoked	NO-AUTH	CANCELED_BY_SERVER	This is sent when deactivating the application in EMRS after it had been used to authenticate in APPLICATION_ONLY mode. Also sent when unchecking the activate checkbox in EMRS for the user after it had been authenticated. Message = “Administrative Action”

9.2.4. //BLP/MKTDATA AND //BLP/MKTVWAP

(SUBSCRIPTION_DATA and SUBSCRIPTION_STATUS Events)

Request	Message Type	Category	Sub-Category	Scenario
Session.subscribe	SubscriptionTerminated	LIMIT		Concurrent subscription limit exceeded.
	SubscriptionTerminated	UNCLASSIFIED		“Failed to obtain initial paint” If this error occurs, the Bloomberg Data Center was unable to get Initial Paint for subscription. User will still receive subscription ticks.
	SubscriptionTerminated	CANCELED		Subscription has been cancelled via Unsubscribe() or Cancel() call.
	SubscriptionStarted			Subscription started.
	SubscriptionStreamsActivated			Data stream successfully opened from a particular host
	SubscriptionStreamsDeactivated			Data stream closed
	SubscriptionFailure	NOT_ENTITLED	EID_NEEDED	Invalid user or credentials or user being blocked by metering server.
	SubscriptionFailure	BAD_TOPIC		Bad Topic string or Service name in Topic
	SubscriptionFailure	BAD_SEC		Bad Security
	SubscriptionFailure	NOT_MONITORABLE		Not a real-time security (no streamId or monid)
	SubscriptionFailure	NOT_APPLICABLE		Field not valid to the specified security
	SubscriptionFailure	BAD_FLD		Invalid field
	SubscriptionFailure	TIMEOUT		Request timed-out
	SubscriptionFailure	UNCLASSIFIED		Invalid field. No permissions
	SubscriptionFailure	SVC_UNAVAILABLE		Contact Bloomberg Help Desk
SubscriptionFailure	NOT_MONITORABLE		No price available or no permission for the specified PCS	

Message Type	Category	Sub-Category	Scenario
TokenGenerationSuccess	N/A	N/A	A token was successfully generated.
TokenGenerationFailure	NO_AUTH	INTERNAL_ERROR	Library or backend errors
TokenGenerationFailure	NO_AUTH	INVALID_USER	User not found in the EMRS database.
TokenGenerationFailure	NO_AUTH	INVALID_APP	Application name not found in the EMRS database.
TokenGenerationFailure	NO_AUTH	CROSS_FIRM_AUTH	Firm number mismatches with user(s) or application(s).
TokenGenerationSuccess			A token was successfully generated.
TokenGenerationFailure	BAD_ARGS	INVALID_USER or INVALID_APP	A token was not generated.

9.2.5. //BLP/REFDATA

(REQUEST_STATUS, RESPONSE and PARTIAL_RESPONSE Events)

Request	Message Type	Category	Sub-Category	Scenario
For All Requests	ResponseError	LIMIT	DAILY_LIMIT_REACHED	Daily limit for user reached
	ResponseError	LIMIT	MONTHLY_LIMIT_REACHED	Monthly limit for user reached
	ResponseError	LIMIT	MANUALLY_DISABLED	Manually disabled user
	ResponseError	LIMIT	FREE_TRIAL_TERM_LIMIT_REACHED	FTT limit reached
	ResponseError	NO_AUTH	INVALID_USER	Invalid ASID or user
	ResponseError	NO_AUTH	NO_PRODUCTS_FOUND	No products found (SAPI only)
	ResponseError	NO_AUTH	CROSS_FIRM_AUTH	User logged in from different firm
	ResponseError	NOT_ENTITLED	NOT_ENTITLED_FIELD	Invalid entitlements for security or field requested
	ResponseError	BAD_SEC	INVALID_SECURITY_IDENTIFIER	Invalid security requested
	ResponseError	UNCLASSIFIED	UNKNOWN	Internal error
	RequestFailure	UNCLASSIFIED		Request Timeout (REQUEST_STATUS)
HistoricalDataRequest	ResponseError	BAD_ARGS	INVALID_START_END	Invalid start/end date requested
	ResponseError	BAD_ARGS	INVALID_CURRENCY	Invalid currency requested
	ResponseError	BAD_ARGS	NO_FIELDS	No fields requested
	ResponseError	BAD_ARGS	TOO_MANY_FIELDS	Requested too many fields
	ResponseError	BAD_FLD	INVALID_FIELD	Invalid field
	ResponseError	BAD_FLD	INVALID_OVERRIDE_FIELD	Invalid override field requested
	ResponseError	BAD_FLD	NOT_APPLICABLE_TO_HIST_DATA	Not valid historical field requested
	ResponseError	BAD_FLD	NOT_APPLICABLE_TO_SECTOR	Historical field not applicable to market sector
	ResponseError	NOT_AVAILABLE	NOT_AVAILABLE_API	No data currently available
IntradayBarRequest	ResponseError	BAD_ARGS	NO_EVENT_TYPE	No Event type requested
IntradayTickRequest	ResponseError	BAD_ARGS	NO_EVENT_TYPE	No Event type requested
ReferenceDataRequest	ResponseError	NOT_AVAILABLE	INVALID_FIELD_DATA	Invalid field
	ResponseError	BAD_ARGS	TOO_MANY_OVERRIDES	Too many override fields requested
	ResponseError	BAD_FLD	INVALID_OVERRIDE_FIELD	Invalid override field requested
	ResponseError	BAD_FLD	NOT_APPLICABLE_TO_REF_DATA	No valid refdata field requested
	fieldExceptions	NO_AUTH	FIELD_NOT_ALLOWED_FOR_DATAFEED_USER	Field not permitted to datafeed users (B-PIPE only). Could pertain to one of the following:

				<p>1) Field is a premium field and only available to BPS users/applications.</p> <p>2) Field not available to any B-PIPE users</p> <p>3) Field is a premium field and B-PIPE license is set up as type 9 with “static data” turned off. Both BPS and Non-BPS users/applications are not allowed to request this field.</p>
IntradayBarRequest	ResponseError	BAD_ARGS	NO_EVENT_TYPE	No Event type requested
IntradayTickRequest	ResponseError	BAD_ARGS	NO_EVENT_TYPE	No Event type requested
categorizedFieldSearchRequest	categorizedFieldResponse	UNCLASSIFIED		Contact Bloomberg
	categorizedFieldSearchError	BAD_ARGS		
	categorizedFieldResponse	BAD_FLD		Invalid request/no search string
	categorizedFieldSearchError			
	categorizedFieldSearchError			
fieldInfoRequest	fieldResponse	UNCLASSIFIED		Contact Bloomberg
	fieldResponse.fieldData.field			Some field IDs are invalid
fieldSearchRequest	fieldSearchError			
	fieldResponse.fieldSearchError			Invalid request/invalid field IDs

10. Event Types

For both synchronous and asynchronous Event handlers, the data and status Events along with the Message types returned will be the same.

Below is a list of the various possible Event types that can be returned by the Bloomberg API services and under what circumstance they are returned. Their purpose is to identify the type of Event received so it can be handled properly. The Event type is obtained by using the Event.Type property and Event.EventType enumeration.

SESSION_STATUS

Changes in the state of the Session, such as Session successfully started or Session is terminated.

SERVICE_STATUS

Changes to the status of an active open service request.

SUBSCRIPTION_DATA

Streaming subscription data delivered for active subscription requests.

SUBSCRIPTION_STATUS

Changes in the state of a pending/active subscription request.

PARTIAL_RESPONSE AND RESPONSE

Both of these Event types are associated with non-subscription data requests. If the data required by the Bloomberg Data Center to fill a request is too large, then Bloomberg will split up the response into multiple Event Messages of type PARTIAL_RESPONSE and RESPONSE. Therefore, each non-final Event Message received by the application will be of type PARTIAL_RESPONSE, while the final Event Message will be of type RESPONSE and will be the responsibility of the application to handle these Event Messages. In other words, only when the RESPONSE Event Message is received should the application stop waiting for more Messages and assume that all of the data for that request has been received.

REQUEST_STATUS

Changes in the state of a pending non-subscription request.

AUTHORIZATION_STATUS

Changes made to the state of an active authorization, such as entitlement changes.

ADMIN

Administrative Events from Bloomberg Services.

An Event Message is then further defined by its Message type, which is covered next.

10.1. MESSAGE TYPES

An Event Message will contain one or more Messages. Each will have a type associated with it—obtained by using the Message.MessageType property. Their purpose is to provide more detail about the Event and whether its action was successful or not. Following are a few such Message types and the Event types they are associated with.

SESSION_STATUS

- **SessionStarted** — Session has been started successfully.
- **SessionTerminated** — Session has been terminated.
- **SessionStartupFailure** — Session has failed to start.
- **SessionConnectionUp** — Session has successfully started for the first time (SessionOptions.AutoRestartOnDisconnect() is set to true).
- **SessionConnectionDown** — Session has failed to start (SessionOptions.AutoRestartOnDisconnect() is set to true).

- **SessionClusterInfo** - Delivered when the SDK is connected to a cluster and receives initial cluster configuration information from the Bloomberg infrastructure. The message includes the name of the cluster and a list of end points within.
- **SessionClusterUpdate** - This is delivered when the SDK is connected to a cluster and receives updated configuration information about the cluster from the Bloomberg infrastructure. The message includes the name of the cluster and lists of end points that have been added/removed.

SERVICE_STATUS

- **ServiceOpened** — Service has been opened successfully.
- **ServiceOpenedFailure** — Service has failed to open.

SUBSCRIPTION_DATA

- **MarketDataEvents** — Message contains market data. A list of possible types of market data (i.e., MKTDATA_EVENT_TYPE/MKTDATA_EVENT_SUBTYPE) will be covered later in this course in the section “Understanding Market Data Event Types and Sub-Types.”

SUBSCRIPTION_STATUS

- **SubscriptionStarted** — Subscription has started.
- **SubscriptionFailed** — Subscription has failed. Reason will be determined by its category and description properties. For instance, if the field mnemonic in the subscription doesn't exist, then the category will be BAD_FLD and the description will be “Unknown Field.”
- **SubscriptionTerminated** — Failed to obtain initial paint or concurrent subscription limit has been reached.
- **SubscriptionStreamActivated** — Delivered when a data set of a subscription has been successfully opened. Data associated with this data set will start ticking afterwards. The message includes a list of data sets that become activated and a reason for the activation.
- **SubscriptionStreamDeactivated** — delivered when the SDK loses its connectivity to the end point providing an active data set. Data associated with this data set will stop ticking afterwards. The message includes a list of data sets that become deactivated and a reason for the deactivation.

PARTIAL_RESPONSE AND RESPONSE

- **ReferenceDataResponse** — Message either contains data (with no errors or exceptions) or contains one of the following elements (in which case, more details available from its category, subcategory and Message):
 - a) **responseError** — For example, if no field was included in the request. In that case, the category will be BAD_ARGS and the Message will be “No fields specified.”
 - b) **securityError** – For example, if the security in the reference data request is invalid, then the category will be BAD_SEC, the subcategory will be INVALID_SECURITY and the Message will be “Unknown/Invalid Security” .
 - c) **fieldExceptions** – For example, if the field in the reference data request is not found in the API Data Dictionary, then the category will be BAD_FLD, the subcategory will be INVALID_FIELD and the Message will be “Field not valid”.

REQUEST_STATUS

- **RequestFailure** — Occurs when the request times out on the Bloomberg backend.

AUTHORIZATION_STATUS

- **AuthorizationSuccess** — User was authorized successfully.
- **AuthorizationFailure** — User was not authorized successfully. Reason determined by looking at the Message's category, subcategory and properties.
- **ResponseError** — Cause of this error will be known only after looking at Message's category, subcategory and properties.

- **EntitlementChanged** — User has logged off and then back on to the Bloomberg Professional service.
- **AuthorizationRevoked** — Reasons could be that either the user logged in to a Bloomberg Professional service other than the one on the PC on which he is running his application, or the user is locked out of the Bloomberg Professional service.

ADMIN

- **SlowConsumerWarning** — Receiving the Message indicates client is slow. NO category/subcategory.
- **SlowConsumerWarningCleared** — Receiving the Message indicates client is no longer slow. NO category/subcategory.

11. Handling Reference Errors/Exceptions

The response to a ReferenceDataRequest request is an Element named ReferenceDataResponse, an Element object that is a CHOICE of an Element named responseError (sent, for example, if the request was completely invalid or if the service is down) or an array of Element objects named securityData, each containing some of the requested data. The structure of these responses can be obtained from the service schema, but is also conveniently viewed by printing the response in the response Event handler code.

```
ReferenceDataResponse (choice) = {
    securityData[] = {
        securityData = {
            security = AAPL US Equity
            sequenceNumber = 0
            fieldData = {
                PX_LAST = 442.66
                DS002 = APPLE INC
                VWAP_VOLUME = 15428164
            }
        }
    }
}
```

Because the Element ReferenceDataResponse is an array, it allows each response Event to receive data for several of the requested securities. The Bloomberg API may return a series of Message objects, each containing a separate ReferenceDataResponse, within a series of Event objects in response to a request. However, each security requested will appear in only one array entry in only one Message object.

Each element of the securityData array is a SEQUENCE that is also named securityData. Each securityData SEQUENCE contains an assortment of data, including values for the fields specified in the request. The reply corresponding to the improperly named security, “INVALID US Equity,” shows that the number and types of fields in a response can vary between entries.

```

ReferenceDataResponse (choice) = {
  securityData[] = {
    securityData = {
      security = INVALID US Equity
      securityError = {
        source = 100::bbdbs1
        code = 15
        category = BAD_SEC
        message = Unknown/Invalid security [nid:100]
        subcategory = INVALID_SECURITY
      }
      sequenceNumber = 2
      fieldData = {
      }
    }
  }
}

```

This response Message has an Element not previously seen named `securityError`. This Element provides details explaining why data could not be provided for this security.

☞ *Note that sending one unknown security does not invalidate the entire request.*

Just printing the response in the default format is educational, but to perform any real work with the response, the values must be extracted from the received Message and assigned elsewhere for use. The following Event handler demonstrates how to navigate the Element structure of the ReferenceDataResponse.

The Element method of Message provides a handle for navigating the contents of the Message objects using Element methods. If an Element object is an array (e.g., `securitydataArray`), then the `numValues` method provides the number of items in the array.

☞ *Note that the Element class also provides a similarly named method, `numElements` (not used in this example), which returns the number of Element objects in a SEQUENCE.*

```

<C++>
static void handleResponseEvent(Event event) {
  MessageIterator iter = event.messageIterator();
  while (iter.next()) {
    Message message = iter.message();
  }
}

```

```

Element ReferenceDataResponse = message.asElement();
if (ReferenceDataResponse.hasElement("responseError")) {
    // handle error
}

Element securityDataArray = ReferenceDataResponse.getElement(
"securityData");

int numItems = securityDataArray.numValues();
for (int i = 0; i < numItems; ++i) {
    Element securityData = securityDataArray.getValueAsElement(i);
    std::string security = securityData.getElementAsString("security");
    int sequenceNumber = securityData.getElementAsInt32(
"sequenceNumber");
    if (securityData.hasElement("securityError")) {
        Element securityError = securityData.getElement("securityError");
        // handle error
        return;
    } else {
        Element fieldData = securityData.getElement("fieldData");
        double px_last = fieldData.getElementAsFloat64("PX_LAST");
        String ds002 = fieldData.getElementAsString("DS002");
        double vwap_vol = fieldData.getElementAsFloat64("VWAP_VOLUME");
        // Individually output each value
        std::cout << security =" << security << std::endl;
        std::cout << "* sequenceNumber=" << sequenceNumber << std::endl;
        std::cout << "* px_last =" << px_last << std::endl;
        std::cout << "* ds002=" << ds002 << std::endl;
        std::cout << "* vwap_volume =" << vwap_vol << std::endl;
        std::cout << std::endl;
    }
}
}
}
}

```

When stepping through the securityData array, the requested Bloomberg fields are accessed by the name and type (e.g., getElementAsFloat64, getElementAsInt32) as specified in the schema. Once values have been assigned to local variables, they can be used as needed. In this example, they are merely output individually in a distinctive format.

The program output is shown below:

```
security =AAPL US Equity
sequenceNumber=0
px_last =442.66
ds002 =APPLE INC
vwap_volume =15428164
security =IBM US Equity
sequenceNumber=1
px_last =213.30
ds002 =INTL BUSINESS MACHINES CORP
vwap_volume =2885962.0
security =BLAHBLAH US Equity
  securityError = {
    source = 100::bbdbs1
    code = 15
    category = BAD_SEC
    message = Unknown/Invalid security [nid:100]
    subcategory = INVALID_SECURITY
  }
```

The sequenceNumber is provided to allow the ordering of PARTIAL_RESPONSE Events from the reference data service.

12. SDK for BLPAPI


The Bloomberg API Software Development Kits (SDKs) are available for download from the “API Download Center” on {WAPI <GO>} for each supported programming language interface, operating system and API product. Each SDK provides working binary and source-code examples, function reference help, the appropriate API library(s) and release notes (change logs).

SDKs are also available at <http://www.bloomberglabs.com/api/>. For COM Data Control, the library interface (blpapicom2.dll) is distributed as part of the monthly Desktop API software push and not included in its COM Data Control SDK. Additionally, the function reference help is on {WAPI <GO>} and not in the “/help” folder installed with the SDK.

SDK LANGUAGES

The primary API interfaces are C/C++, Java and .NET. The remaining interfaces (COM Data Control, Python API and Perl API) are built on top of the C API.

API Language	Windows	Linux	Solaris
C/C++	✓	✓	✓
.NET	✓		
Java	✓	✓	✓
COM Data Control	✓		
Perl	✓	✓	✓
Python	✓	✓	

 **Note:** The Desktop API is only supported on Microsoft Windows.

API LIBRARIES

The object models of the Java, .NET, and C++ API libraries are identical, while the C interface provides a C-style version of the object model. The C++ libraries are distributed in 64- and 32-bit versions on all platforms, while the .NET/Java API libraries are 32-bit only. However, the Java and .NET libraries are fully compatible with 64-bit applications and the same library version can be used in both 32- and 64-bit applications.

The Perl and Python APIs require the installation of the C++ API SDK. The Windows version of the Perl API is supported only with ActivePerl 5.0, or greater. The Python API is supported with 32- and 64-bit CPython v2.6 and v2.7.

CODE EXAMPLES

The BLPAPI provides a core set of code examples across multiple languages, products and operating systems. These include examples that demonstrate simple requests and subscriptions, along with those providing full authentication, authorization and permissioning, and enhanced error and data handling. Install the entire set of product SDKs for each language (from the “examples” folder of each SDK installation) and understand what is distributed with each. The product will dictate which additional examples are provided and their functionality.

Binary examples for the C++ and Java API SDKs are found in the “bin” folder of each of their SDK installation folders. They accept command-line parameters to customize how the application will authenticate (if applicable) and what data will be requested/subscribed/published.

At this time, the Windows SDKs from WAPI <GO> have the most complete set of examples and example functionality.

FUNCTION REFERENCE HELP

Many of the SDKs include function reference help, which is found in the “doc” folder. The C++ API help is generated by doxygen, the .NET API help by Sandcastle and the Java API help by javadoc. This is also hosted online at <http://www.bloomberglabs.com/api/documentation/>.

API DEMO TOOL

The BLPAPI provides a Windows API Demo Tool for each of the Desktop API, Server API and B-PIPE API products. This is in the “\bin” folder along with the latest .NET API library that it will reference. For example, the Server API Demo Tool is in the “C:\blp\ServerApi\bin.” It was developed using C# and is a WinForm application.

The API Demo Tools provide a core set of functionality, including:

- Connecting to the product’s process server
- Requesting reference, historical, intraday, schema and API field data
- Subscribing to real-time streaming data

The Server API and B-PIPE API Demo Tools also provide applicable authorization/authentication/permissioning features; the B-PIPE API Demo Tool includes additional subscription tabs such as MarketDepth and MarketList.

13. Request/Response

The Request/Response paradigm is useful in obtaining a one-time response that does not require updates. With this paradigm, all data is static as opposed to the real-time/delayed streaming data received when making a subscription-based request.

A static request is made for the most current security data as well as requests for historical end-of-day and intraday data through the following steps:

1. Establish a connection calling the *Session.start* method.
2. Open the “//blp/refdata” service using the *Session.openService* method.
3. Get the “//blp/refdata” service using the *Session.getService* method.
4. Create a Request object of a type matching the type of request. The valid request types are defined in the API schema. For instance, to make a reference data request, create a Request object of type “ReferenceDataRequest”; historical requests would be of type “HistoricalDataRequest.”
5. Add one or more securities and fields to the Request object, making sure that the fields are valid for that type of request and also the Request/Response paradigm. For instance, “LAST_PRICE” is a real-time field, so it would not be used for a reference data request (as indicated on FLDS <GO>). However, all real-time fields do have an equivalent static field and are, therefore, usually valid for any static request. If unsure of a specific field’s availability for the particular security(ies), use the “FLDS <GO>” function on the Bloomberg Professional service or the API Field (“//blp/apiflds”) service, which will be covered later in this module.
6. Set any particular elements applicable to that type of request.
7. Submit the request using the *Session.sendRequest* method.
8. Handle the data and status responses either synchronously or asynchronously; these are returned as Events. Determine whether the data request has been completely filled by checking the Event type of that Event object. If a RESPONSE Event type is received, no more data will be returned for that request. However, if a PARTIAL_RESPONSE Event type is received, this Event contains only a partial response for the user’s request and further incoming Events must be read until a RESPONSE Event type is retrieved.

13.1. REQUESTING HISTORICAL DATA

The HistoricalDataRequest Request object enables the retrieval of end-of-day data for a set of securities and fields over a specified period, which can be set to daily, monthly, quarterly, semiannually or annually. At least one security and one field are required along with start and end dates.

A range of options can be specified in the request; these are outlined in “Reference Services and Schemas Guide.” The example below shows how to construct a HistoricalDataRequest for monthly last price data for 2012:

```
<C++>
std::vector<std::string> d_securities;
std::vector<std::string> d_fields;
d_securities.push_back("IBM US Equity");
d_securities.push_back("GOOG Equity");
d_fields.push_back("PX_LAST");
```



```

d_fields.push_back("PX_BID");
.....
// Assume that the //blp/refdata service has already been opened
Service refDataService = session.getService("//blp/refdata");
Request request = refDataService.createRequest("HistoricalDataRequest");
for(int i = 0; i < (int)d_securities.size(); i++)
{
    request.getElement("securities").appendValue(d_securities[i].c_str());
}
for(int k = 0; k < (int)d_fields.size(); k++)
{
    request.getElement("fields").appendValue(d_fields[k].c_str());
}
request.set("periodicitySelection", "MONTHLY");
request.set("startDate", "20120101");
request.set("endDate", "20121231");
std::cout << "Sending Request: " << request << std::endl;
session.sendRequest(request);

```

13.2. HANDLING HISTORICAL DATA MESSAGES

A successful `HistoricalDataResponse` (with no errors or exceptions) holds information on a single security. It contains a `HistoricalDataTable` with one `HistoricalDataRow` for each interval returned.

```

<C++>
while (true)
{
    Event event = session.nextEvent();
    MessageIterator msgIter(event);
    while (msgIter.next())
    {
        Message &msg = msgIter.message();
        if ((event.eventType() != Event::PARTIAL_RESPONSE) &&
            (event.eventType() != Event::RESPONSE))
        {

```

```

        continue;
    }
    Element securityData = msg.getElement(SEcurity_DATA);
    Element securityName = securityData.getElement(SEcurity_NAME);
    std::cout << securityName << "\n\n";

    //only process field data if no errors or exceptions have occurred
    if(!ProcessExceptions(msg))
    {
        if(!ProcessErrors(msg))
        {
            ProcessFields(msg);
        }
    }
    std::cout << "\n\n";
}
if (event.eventType() == Event::RESPONSE) {
    break;
}
}

```

In the above while() loop, if there are no exceptions or errors, the ProcessFields function is called. To view the code for this function, look at the HistoryExample C++ example, which is found in the Server C++ API SDK installation. Currently, this example is not available in the B-PIPE SDK.

13.3. REQUESTING INTRADAY BAR DATA

Bloomberg maintains a tick-by-tick history going back 140 business days for all securities where streaming data is available. This intraday data can be used to draw detailed charts, for technical analysis or to retrieve the initial data for a monitoring graph function such as the “GIP <GO>” function on the Bloomberg Professional service.

The Intraday Bar Request enables retrieval of summary intervals for intraday data covering five Event types: TRADE, BID, ASK, BEST_BID and BEST_ASK, over a period of time.

☞ *Note that only one event type can be specified per request.*

Each bar contains OPEN, HIGH, LOW, CLOSE, VOLUME and NUMBER_OF_TICKS. The interval size of the bars can be set from 1 minute to 1,440 minutes (24 hours).

Each IntradayBarRequest can only submit one single instrument. In addition, the Event type, interval, and date/time start and end-points in UTC (Coordinated Universal Time) must be specified. The example below shows how to construct an IntradayBarRequest:

```

<C++>
Service refDataService = session.getService("//blp/refdata");
Request request = refDataService.createRequest("IntradayBarRequest");
// only one security/eventType per request
request.set("security", "IBM US Equity");
request.set("eventType", "TRADE");
request.set("interval", 60);
Datetime startDateTime, endDateTime;
getTradingDateRange(&startDateTime, &endDateTime); // Custom function call
request.set("startDateTime", startDateTime);
request.set("endDateTime", endDateTime);
request.set("gapFillInitialBar", true);
std::cout << "Sending Request: " << request << std::endl;
session.sendRequest(request);

```

The code snippet above demonstrates an `IntradayBarRequest` “IBM US Equity” for trade bars with an interval of 60 minutes between the start and end date—time that would be set in the `getTradingDateRange` function.

13.4. HANDLING INTRADAY BAR DATA MESSAGES

A successful `IntradayBarResponse` will contain an array of `BarTickData` each of which contains open, high, low, close, number of Events and volume values.

☞ *For additional information, see “Reference Services and Schemas Guide.”*

The example below shows how to interpret an `IntradayBarResponse`.

```

<C++>
void processMessage(Message &msg) {
    Element data = msg.getElement(BAR_DATA).getElement(BAR_TICK_DATA);
    int numBars = data.numValues();
    std::cout << "Response contains " << numBars << " bars" << std::endl;
    std::cout << "Datetime\t\tOpen\t\tHigh\t\tLow\t\tClose"
    << "\t\tNumEvents\tVolume" << std::endl;
    for (int i = 0; i < numBars; ++i) {
        Element bar = data.getValueAsElement(i);
        Datetime time = bar.getElementAsDatetime(TIME);
        double open = bar.getElementAsFloat64(OPEN);
        double high = bar.getElementAsFloat64(HIGH);

```

```
double low = bar.getElementAsFloat64(LOW);
double close = bar.getElementAsFloat64(CLOSE);
int numEvents = bar.getElementAsInt32(NUM_EVENTS);
long long volume = bar.getElementAsInt64(VOLUME);
std::cout.setf(std::ios::fixed, std::ios::floatfield);
std::cout << time.month() << '/' << time.day() << '/' << time.year()
<< " " << time.hours() << ":" << time.minutes()
<< "\t\t" << std::showpoint
<< std::setprecision(3) << open << "\t\t"
<< high << "\t\t"
<< low << "\t\t"
<< close << "\t\t"
<< numEvents << "\t\t"
<< std::noshowpoint
<< volume << std::endl;
    }
}
```

13.5. REQUESTING INTRADAY TICK DATA

Bloomberg maintains a tick-by-tick history going back 140 business days for all securities where streaming data is available. This intraday data can be used to draw detailed charts, for technical analysis purposes, or to retrieve the initial data for a monitoring graph function (such as the “GIP <GO>” function).

The `IntradayTickRequest` enables retrieval of tick-by-tick history for a single security. In addition, the Event type(s), interval and date/time start and end points in UTC (Coordinated Universal Time) must be specified. The example below shows how to construct an `IntradayTickRequest`:

```
<C++>
Service refDataService = session.getService("//blp/refdata");
Request request = refDataService.createRequest("IntradayTickRequest");
request.set("security", "VOD LN Equity");
request.append("eventTypes", "TRADE");
request.append("eventTypes", "AT_TRADE");
Datetime startDateTime, endDateTime ;
getTradingDateRange(&startDateTime, &endDateTime) / Custom Function
request.set("startDateTime", startDateTime);
request.set("endDateTime", endDateTime);
request.set("includeConditionCodes", true);
std::cout << "Sending Request: " << request << std::endl;
session.sendRequest(request);
```

In the above code, an intraday tick request is being made for VOD LN Equity for both TRADE and AT_TRADE tick data and condition codes from the start and end date/times that are returned by the custom function `getTradingDateRange`.

Several `eventTypes` are valid for an intraday tick request. They are defined in the API schema for the `IntradayTickRequest` type. Users may include as many as they like in one request.

Following is the current list of enumeration `eventTypes`:

- TRADE
- BID
- ASK
- BID_BEST
- ASK_BEST
- BID_YIELD
- ASK_YIELD
- MID_PRICE
- AT_TRADE
- BEST_BID

- BEST_ASK
- SETTLE

13.6. HANDLING INTRADAY TICK DATA MESSAGES

A successful `IntradayTickResponse` will contain an array of `IntradayTickData` providing information on each tick in the specified time range. The time to respond to this request is influenced by the date and time range of the request and the level of market activity during that period.

```
<C++>
void processMessage(Message &msg)
{
    Element data = msg.getElement("tickData").getElement("tickData");
    int numItems = data.numValues();
    std::cout << "TIME\t\t\t\tTYPE\tVALUE\t\tSIZE\tCC" << std::endl;
    std::cout << "----\t\t\t\t----\t-----\t\t----\t--" << std::endl;
    std::string cc;
    std::string type;
    for (int i = 0; i < numItems; ++i) {
        Element item = data.getValueAsElement(i);
        Datetime time = item.getElementAsDatetime("time");
        std::string timeString = item.getElementAsString("time");
        type = item.getElementAsString("type");
        double value = item.getElementAsFloat64("value");
        int size = item.getElementAsInt32("size");
        if (item.hasElement("conditionCodes")) {
            cc = item.getElementAsString("conditionCodes");
        } else {
            cc.clear();
        }
        std::cout.setf(std::ios::fixed, std::ios::floatfield);
        std::cout << timeString << "\t"
<< type << "\t"
<< std::setprecision(3)
<< std::showpoint << value << "\t\t"
<< size << "\t" << std::noshowpoint
<< cc << std::endl;
```

```

}
}

```

The named constants were replaced by their explicit element strings for clarity. However, for best practices, define these element strings using the `Name` class as this is more efficient and less prone to errors.

13.7. SENDREQUEST METHOD

The `sendRequest` method is a member of the `Session` class and is used to submit any of the aforementioned static data requests, as well as other static data requests that have not been covered. There are two overloads for this method in the C++ API library. In addition to the required `Request` object parameter, it also provides the ability to pass optional arguments. These include a `CorrelationId`, `EventQueue`, `Identity` and `requestLabel/requestLabelLen` values.

The two method overload definitions are:

```

<C++>
void blpapi::Session::sendRequest(const Request& request,
                                const CorrelationId& correlationId = CorrelationId(),
                                EventQueue* eventQueue=0,
                                const char* requestLabel=0,
                                int requestLabelLen=0)

void blpapi::Session::sendRequest (const Request& request,
                                   const Identity& user,
                                   const CorrelationId& correlationId = CorrelationId(),
                                   EventQueue* eventQueue=0,
                                   const char* requestLabel=0,
                                   int requestLabelLen=0)

```

In the first overload, send the specified `Request` object; in the second overload; send the specified request using the specified `Identity` for authorization.

If the optionally specified `CorrelationID` is supplied, use it, otherwise create a `CorrelationID`. The actual `CorrelationID` used is returned. If the optionally specified `EventQueue` is supplied, all `Events` relating to this request will arrive on that `EventQueue`. If the optional `requestLabel` and `requestLabelLen` are provided, they define a string that will be recorded along with any diagnostics for this operation. There must be at least `requestLabelLen` printable characters at the location `requestLabel`.

A successful request will generate zero or more `PARTIAL_RESPONSE` Messages followed by exactly one `RESPONSE` Message. Once the final `RESPONSE` Message has been received, the `CorrelationID` associated with this request may be reused. If the request fails at any stage, a `REQUEST_STATUS` will be generated after which the `CorrelationID` associated with the request may be reused.

14. Multi-Threading and the API

The `Session` class provides two ways of handling Events. The simpler of the two is to call the `nextEvent` method to obtain the next available Event object. This method will block until an Event becomes available and is well-suited for single-threaded customer applications. This is known as “synchronous mode.”

For an asynchronous mode application, an `EventHandler` object is supplied when creating a `Session`. In this case, the user-defined `processEvent` method in the supplied `EventHandler` is called by the Bloomberg API when an Event is available. The signature for the `processEvent` method is:

```
public void processEvent(Event event, Session session)
// Note: no exceptions are thrown
```

The calls to the `processEvent` method will be executed by a thread owned by the Bloomberg API, making the customer application multi-threaded. Consequently, a customer application must in this case ensure that data structures and code accessed from both its main thread and from the thread running the `EventHandler` object are thread-safe.

Most applications will only use a single `Session`. However, the Bloomberg API allows the creation of multiple `Session` objects. Multiple instances of the `Session` class do not contend and thus allow for efficient multi-threading.

There is no reason to create additional threads as the Bloomberg API already utilizes multiple threads where deemed necessary to provide the most efficient handling of connections and data responses.

To implement a worker thread that would perform any processing required once subscription-based tick data has been received on the Bloomberg API-owned thread, users must ensure that all of the ticks retain their order (i.e., sequence). However, the option to create a thread model is available and it may be necessary under certain circumstances, for example, when subscribing to a large volume of active instruments with a lot of processing/analysis performed on those ticks.

For request/response type data, notice that the `Session.sendRequest` method accepts an `EventQueue`. This allows the user to specify an Event handler for that particular request. This allows the handling of static data requests on a separate thread, while a subscription-based request is being handled on the supplied Event handler when the `Session` object was created.

`EventDispatchers` allow dispatching Events from multiple threads. If `EventDispatcher` is not specified, a default one is created for the `Session`. `EventDispatchers` can be shared across Event queues, thus a pool of threads can be created to dispatch Events from multiple Event queues. To use an `EventDispatcher`, create an `EventDispatcher` with the number of threads needed for dispatching Events and pass to the `Session` at construction time. The `Session` will then use this `EventDispatcher` for dispatching Events.

14.1. COMBINING REFERENCE AND SUBSCRIPTION DATA

When developing an application that will handle real-time streaming and static data, a separate `Session` can be used for each type of data. The reason: to ensure that the processing of a heavyweight subscription, for instance, is not being slowed down by the reading and blocking of multiple static request responses. In fact, there may be a separate `Session` for a subscription, another for frequent reference data requests (for fields that are not available in a real-time format) and another for occasional large intraday-type requests.

15. Converting from Excel Formulas (TransExcelFormToCOM)

15.1. BDP(): STREAMING DATA (REAL-TIME OR DELAYED).

One use of the BDP() function is to subscribe to streaming data for a single security and real-time field. In the v3 API, to subscribe to a security and real-time field and retrieve updates as they occur at the source, the Subscription Paradigm is required.

Service: “//blp/mktdata”

BDP Parameters	Description
Update Frequency: This is set in Excel through the Bloomberg “Options” dialog box in the Bloomberg Excel ribbon (under the “Functions” tab).	Update frequency enables users to set, in milliseconds, how often they receive ticks from the Bloomberg Data Center. The default is 300 milliseconds.
Sample Code Translation	
<p><u>Bloomberg Excel Formula:</u></p> <pre>BDP("IBM US Equity", "LAST_PRICE")</pre> <p><u>Equivalent COM v3 Data Control Subscription Code:</u></p> <pre>Dim WithEvents session As blpapicomLib.session Set session = New blpapicomLib2.session session.QueueEvents = True session.Start session.OpenService "//blp/mktdata" Dim sub As blpapicomLib2.SubscriptionList Dim cid As blpapicomLib2.CorrelationId Set sub = session.CreateSubscriptionList() Set cid = session.CreateCorrelationId(1) sub.Add "IBM US Equity", "LAST_PRICE", cid ' To set an update interval of, for e.g. 4 seconds, ' use this syntax instead: ' sub.AddEx "IBM US Equity", "LAST_PRICE", "interval=4.0" session.Subscribe sub</pre>	

15.2. BDP(): REFERENCE DATA (STATIC)

The other use of the BDP() function is to request reference data (a one-time snapshot of data) for a single security and static field. In the v3 API, to request reference data, the Request/Response Paradigm is required.

Service: “//blp/refdata”

Request Type: ReferenceDataRequest

COM v3 Data Control Resources:

BDP Parameters	Description
Specifying Overrides	Calculation Overrides
Sample Code Translation (click to show/hide)	

Bloomberg Excel Formula:

```
BDP("IBM US Equity","EQY_WEIGHTED_AVG_PX","VWAP_START_TIME=9:30")
```

Equivalent COM v3 Data Control Request Code:

```
Dim WithEvents session As blpapicomLib2.session
Dim refdataservice As blpapicomLib2.Service

Set session = New blpapicomLib2.session
session.QueueEvents = True
session.Start
session.OpenService "//blp/refdata"

Set refdataservice = session.GetService("//blp/refdata")
Dim req As REQUEST
Set req = refdataservice.CreateRequest("ReferenceDataRequest")
req.GetElement("securities").AppendValue "IBM US Equity"
req.GetElement("fields").AppendValue "EQY_WEIGHTED_AVG_PX"
Dim overrides As Element
Set overrides = req.GetElement("overrides")
Dim override As Element
Set override = overrides.AppendElement()
override.SetElement "fieldId", "VWAP_START_TIME"
override.SetElement "value", "9:30"
session.SendRequest req
```

15.3. BDS(): BULK DATA (STATIC)

The BDS() function requests bulk reference data for a single security and bulk field. In the API, there is no difference in the request type and options for reference data and bulk reference data (the field dictates if it is bulk). The difference between the two lies in parsing the response—bulk data responses are returned in a different format.

Service: "//blp/refdata"

Request Type: ReferenceData

BDP Parameters	Description
Specifying Overrides	Calculation Overrides
Sample Code Translation (click to show/hide)	

```

Bloomberg Excel Formula:
BDS("IBM US Equity", "OPT_CHAIN")
Equivalent COM v3 Data Control Request Code:
Dim WithEvents session As blpapicomLib2.session
Dim refdataservice As blpapicomLib2.Service

Set session = New blpapicomLib2.session

session.QueueEvents = True
session.Start

session.OpenService "//blp/refdata"
Set refdataservice = session.GetService("//blp/refdata")

Dim req As REQUEST
Set req = refdataservice.CreateRequest("ReferenceDataRequest")

req.GetElement("securities").AppendValue "IBM US Equity"
req.GetElement("fields").AppendValue "OPT_CHAIN"

session.SendRequest req
    
```

15.4. BDH(): HISTORICAL “END-OF-DAY” DATA (STATIC)

One use of the BDH() function is to request historical “end-of-day” data for a single security and historical field. The Request/Response Paradigm is required with the reference data service.

Service: “//blp/refdata”

Request Type: HistoricalDataRequest

BDP Parameters	Description	API Programming Equivalent * Denotes default value
Specifying Overrides	Calculation overrides	
Start Date	The start date for the historical request.	Element: startDate Element value (string): yyyyymmdd
End Date	The end date for the historical request	Element: endDate Element value (string): yyyyymmdd
Calendar Codes (CDR)	Returns the data based on the calendar of the specified country, exchange or region from CDR <GO>. It's arguments are a two-character calendar code null terminated string. This will cause the data to be aligned according to the calendar, including calendar holidays. This applies only to DAILY requests.	Element: calendarCodeOverride Element value (string): CDR<GO> Calendar Type
Currency (FX)	Amends the value from local currency of the security to the desired currency.	Element: currency Element value (string): Currency of the ISO code. e.g., USD, GBP

BDP Parameters	Description	API Programming Equivalent * Denotes default value
Date Format (DtFmt)	Defines the date format as either regular or relative. Set to True for relative.	Element: returnRelativeDate Element value: TRUE or FALSE
Days	Sets whether to include or exclude Non-Trading Days when no data is available.	Element: nonTradingDayFillOption Element value (string): NON_TRADING_WEEKDAYS*, ALL_CALENDAR_DAYS or ACTIVE_DAYS_ONLY
Fill	Formats the type of data returned for non-trading days.	Element: nonTradingDayFillMethod Element value (string): PREVIOUS_VALUE* or NIL_VALUE
Period (Per)	Sets the periodicity of the data	Element: periodicityAdjustment Element value (string): ACTUAL, CALENDAR* or FISCAL Element: periodicitySelection Element value (string): DAILY*, WEEKLY, MONTHLY, QUARTERLY, SEMI_ANNUALLY or YEARLY
Points	The number of periods to download from the end date. The response will contain up to X data points, where X is the integer specified. If the original data set is larger than X, the response will be a subset containing the last X data points. Hence, the first range of data points will be removed.	Element: maxDataPoints Element value: Any positive integer
Quote Type (QtTyp)	Sets quote to Price or Yield for a debt instrument.	Element: pricingOption Element value (string): PRICING_OPTION_PRICE or PRICING_OPTION_YIELD*
Quote	Indicates whether to use the average or closing price	Element: overrideOption Element value (string): OVERRIDE_OPTION_CLOSE* or OVERRIDE_OPTION_GPA
Use DPDF Settings (UseDPDF)	Setting to TRUE will follow the DPDF <GO> Terminal function. Here are the rules for this element: If set to TRUE (default), then the DPDF <GO> setting is followed and adjustmentSplit, adjustmentAbnormal and adjustmentNormal are ignored. If set to FALSE, then users will follow whatever setting is specified for adjustmentSplit, adjustmentAbnormal and adjustmentNormal. If nothing is specified, then no adjustment takes place.	Element: adjustmentFollowDPDF Element value: TRUE* or FALSE

BDP Parameters	Description	API Programming Equivalent * Denotes default value
Cash Adjustment Abnormal (CshAdjAbnormal)	Adjust historical pricing to reflect: Special Cash, Liquidation, Capital Gains, Long-Term Capital Gains, Short-Term Capital Gains, Memorial, Return of Capital, Rights Redemption, Miscellaneous, Return Premium, Preferred Rights Redemption, Proceeds/Rights, Proceeds/Shares, Proceeds/Warrants	Element: adjustmentAbnormal Element value: TRUE or FALSE
Capital Changes (CapChg)	Adjust historical pricing and/or volume to reflect: Spin-Offs, Stock Splits/Consolidations, Stock Dividend/Bonus, Rights Offerings/Entitlement	Element: adjustmentSplit Element value: TRUE or FALSE
Cash Adjustment Normal (CshAdjNormal)	Adjust historical pricing to reflect: Regular Cash, Interim, 1st Interim, 2nd Interim, 3rd Interim, 4th Interim, 5th Interim, Income, Estimated, Partnership Distribution, Final, Interest on Capital, Distribution, Prorated	Element: adjustmentNormal Element value: TRUE or FALSE

Sample Code Translation (click to show/hide)

Bloomberg Excel Formula (*):

```
BDH("IBM US Equity","PX_LAST","CQ22009","CQ22010",
"Sort=A","Quote=C","QtTyp=Y","Days=A","Per=cd",
"DtFmt=D","Fill=P","UseDPDF=Y")
```

Equivalent COM v3 Data Control Request Code:

```
Dim WithEvents session As blpapicomLib2.Session
Dim refdataservice As blpapicomLib2.service
Dim req As blpapicomLib2.Request
```

```
Set session = New blpapicomLib2.session
session.Start
session.OpenService "//blp/refdata"
```

```
Set refdataservice = session.GetService("//blp/refdata")
Set req = refdataservice.CreateRequest("HistoricalDataRequest")
```

```
req.GetElement("securities").AppendValue ("IBM US Equity")
req.GetElement("fields").AppendValue ("PX_LAST")
req.Set "returnRelativeDate", "TRUE"
req.Set "periodicityAdjustment", "CALENDAR"
req.Set "periodicitySelection", "DAILY"
req.Set "startDate", "CQ22009"
req.Set "endDate", "CQ22010"
req.Set "nonTradingDayFillMethod", "PREVIOUS_VALUE"
req.Set "nonTradingDayFillOption", "ALL_CALENDAR_DAYS"
req.Set "pricingOption", "PRICING_OPTION_YIELD"
req.Set "overrideOption", "OVERRIDE_OPTION_CLOSE"
req.Set "adjustmentFollowDPDF", "TRUE"
```

```
session.SendRequest req
```

* Some parameters in the BDH() function, such as "Sort", are unavailable in the API schema, as they are unique to the Bloomberg Excel add-ins.

15.5. BDH(): INTRADAY TICKS (STATIC)

Another use of the BDH() function is to request intraday raw ticks for a single security and Event type. The Request/Response Paradigm is required with the reference data service.

Service: “//blp/refdata”

Request Type: IntradayTickRequest

BDP Parameters	Description	API Programming Equivalent
Fields	List of the fields to specify	Element: eventType Element value: TRADE, BID, ASK, BID_BEST, BEST_BID, ASK_BEST, BEST_ASK, MID_PRICE or AT_TRADE
Start Date and Time	Start date and time for the intraday tick request	Element: startDateTime Element value (string): yyyy-mm-dd Thh:mm:ss
End Date and Time	End date and time for the intraday tick request	Element: endDateTime Element value (string): yyyy-mm-dd Thh:mm:ss
IntradayRaw (IntRw)	Indicates an Intraday Tick Request if set to True. This is the default.	Indicated by the request type when creating the request. In this case, the request type would be IntradayTickRequest.
ConditionCodes(CondCodes)	Shows the condition codes of a trade	Element: includeConditionCodes Element value: TRUE (show the data) or FALSE (hide the data)
ExchangeCodes (ExchCode)	Returns the exchange code of the trade	Element: includeExchangeCodes Element value: TRUE (show the data) or FALSE (hide the data)
Show QRM Equivalent(QRM)	Returns all ticks, including those with condition codes. Allows retrieving full QRM ticks if TRUE or standard API subset if FALSE (default value).	Element: includeNonPlottableEvents Element value: TRUE (show the data) or FALSE (hide the data)
Broker Codes (BrkrCodes)	The broker code for Canadian, Finnish, Mexican, Philippine and Swedish equities only. The Market Maker Lookup screen, MMTK <GO>, displays further information on market makers and their corresponding codes.	Element: includeBrokerCodes Element value: TRUE (show the data) or FALSE (hide the data)
TRAC RPS Codes (RPSCodes)	The Reporting Party Side. The following values appear: - B: Customer transaction where the dealer purchases securities from the customer. - S: Customer transaction where the dealer sells securities to the customer. - D: Inter-dealer transaction (always from the sell side).	Element: includeRpsCodes Element value: TRUE (show the data) or FALSE (hide the data)
Trade Time (TradeTime)	Display the time of the trade.	Element: includeTradeTime Element value: TRUE (show the data) or FALSE (hide the data)
Action Codes (ActionCodes)	Displays additional information about the action associated with the trade.	Element: includeActionCodes Element value: TRUE (show the data) or FALSE (hide the data)

BDP Parameters	Description	API Programming Equivalent
Yield (Yield)	Displays the yield for the trade.	Element: includeYield Element value: TRUE (show the data) or FALSE (hide the data)
Spread Price (Spread)	Displays the spread price.	Element: includeSpreadPrice Element value: TRUE (show the data) or FALSE (hide the data)
Upfront Price (UpfrontPrice)	Displays the pricing for credit default swaps (CDS).	Element: includeUpfrontPrice Element value: TRUE (show the data) or FALSE (hide the data)
Indicator Codes (IndicatorCodes)	Displays additional indicator information about the trade.	Element: includeIndicatorCodes Element value: TRUE (show the data) or FALSE (hide the data)
Sample Code Translation (click to show/hide)		

BDP Parameters	Description	API Programming Equivalent
Bloomberg Excel Formula (*) (**):	BDH("IBM US Equity", "ASK,BID,TRADE", "5/27/2010 9:00:00 AM", "", "Dir=V", "Dts=S", "Sort=A", "IntrRw=True", "CondCodes=S", "QRM=S", "ExchCode=S", "BrkrCodes=S", "RPSCodes=S")	<u>Equivalent COM v3 Data Control Request Code:</u>
	Dim WithEvents session As blpapiComLib2.Session Dim refdataService As blpapiComLib2.service Dim req As blpapiComLib2.Request Dim startTime As blpapiComLib2.DateTime Dim endTime As blpapiComLib2.DateTime	
	Set session = New blpapiComLib2.session session.Start session.OpenService "//blp/refdata"	
	Set refdataService = session.GetService("//blp/refdata") Set req = refdataService.CreateRequest("IntradayTickRequest")	
	req.Append "eventTypes", "TRADE" req.Append "eventTypes", "ASK" req.Append "eventTypes", "BID"	
	' All times are in GMT	
	Set startTime = session.CreateDatetime startTime.Year = 2010 startTime.Month = 5 startTime.Day = 27 startTime.Hour = 14 startTime.Minute = 30 startTime.Second = 0 startTime.Millisecond = 0	
	req.Set "startDateTime", startTime	
	req.Set "includeExchangeCodes", "TRUE" req.Set "includeConditionCodes", "TRUE" req.Set "includeBrokerCodes", "TRUE" req.Set "includeRpsCodes", "TRUE" req.Set "includePlottableEvents", "TRUE"	
	session.SendRequest(req)	
	* All times in a BDH() intraday calls are in local time, whereas in the v3 API they are in GMT. The above BDH() time is assumed to be GMT-5:00 (EST)	
	** Some parameters in the BDH() function, such as "Dir" are unavailable in the API schema as they are unique to the Bloomberg Excel add-ins.	

15.6. BDH()/BRB(): INTRADAY BAR DATA (STATIC/SUBSCRIPTION)

Static-based: The third use of the BDH() function is to request static intraday bar data for a single security and Event type. The Request/Response Paradigm is required with the reference data service.

Service: “//blp/refdata”

Request Type: IntradayBarRequest

BDP Parameters	Description	API Programming Equivalent
Bar Type (BarTp)	Defines the market event supplied for an Intraday request.	Element: eventType Element value: TRADE, BID or ASK
Bar Size (BarSz)	Sets the length of each time-bar in the response. Entered as a whole number (between 1 and 1,440 minutes). If omitted, the request will default to 1 minute. One minute is the lowest possible granularity.	Element: interval Element value: Whole number between 1 and 1,440 (default = 1)
Start Date and Time	Start date and time for the Intraday Bar Request	Element: startDateTime Element value (string): yyyy-mm-dd Thh:mm:ss
End Date and Time	End date and time for the Intraday Bar Request	Element: endDateTime Element value (string): yyyy-mm-dd Thh:mm:ss
Intraday Raw (IntRw)	Indicates an Intraday Bar Request if set to False.	This is indicated by the request type when creating the request. In this case, the request type would be IntradayBarRequest.
Capital Changes (CapChg)	Adjust historical pricing and/or volume to reflect: Spin-Offs, Stock Splits/Consolidations, Stock Dividend/Bonus, Rights Offerings/Entitlement.	Element: adjustmentSplit Element value: TRUE or FALSE
Cash Adjustment Abnormal (CshAdjAbnormal)	Adjust historical pricing to reflect: Special Cash, Liquidation, Capital Gains, Long-Term Capital Gains, Short-Term Capital Gains, Memorial, Return of Capital, Rights Redemption, Miscellaneous, Return Premium, Preferred Rights Redemption, Proceeds/Rights, Proceeds/Shares, Proceeds/Warrants	Element: adjustmentAbnormal Element value: TRUE or FALSE
Cash Adjustment Normal (CshAdjNormal)	Adjust historical pricing to reflect: Regular Cash, Interim, 1st Interim, 2nd Interim, 3rd Interim, 4th Interim, 5th Interim, Income, Estimated, Partnership Distribution, Final, Interest on Capital, Distribution, Prorated	Element: adjustmentNormal Element value: TRUE or FALSE
Use DPDF Settings (UseDPDF)	Setting to True will follow the DPDF <GO> Terminal function. True is the default setting for this option.	Element: adjustmentFollowDPDF Element value: TRUE or FALSE
Sample Code Translation (static-based)		

BDP Parameters	Description	API Programming Equivalent
Bloomberg Excel Formula (*) (**):		
BDH("IBM US Equity", "OPEN,HIGH,LOW,LAST_PRICE,VOLUME",		
"6/21/2010 9:30:00AM", "", "BarTp=T", "BarSz=30", "Dir=V",		
"Dts=S", "Sort=A", "Quote=C", "UseDPDF=Y")		
Equivalent COM v3 Data Control Request Code:		
Dim WithEvents session As blpapicomLib2.Session		
Dim refdataservice As blpapicomLib2.service		
Dim req As blpapicomLib2.Request		
Dim startTime As blpapicomLib2.DateTime		
Dim endTime As blpapicomLib2.DateTime		
Set session = New blpapicomLib2.session		
session.Start		
session.OpenService "//blp/refdata"		
Set refdataservice = session.GetService("//blp/refdata")		
Set req = refdataservice.CreateRequest("IntradayBarRequest")		
req.Set "security", "IBM US Equity"		
req.Set "interval", 30 ' Set for 30 minute bars		
req.Set "eventType", "TRADE"		
' All times are in GMT		
Set startTime = session.CreateDatetime		
startTime.Year = 2010		
startTime.Month = 5		
startTime.Day = 27		
startTime.Hour = 14		
startTime.Minute = 30		
startTime.Second = 0		
startTime.Millisecond = 0		
req.Set "startDateTime", startTime		
req.Set "adjustmentFollowDPDF", "TRUE"		
session.SendRequest req		
* All times in a BDH() intraday call are in local time,		
whereas in the v3 API they are in GMT. The above BDH()		
time is assumed to be GMT-5:00 (EST)		
** Some parameters in the BDH() function, such as "Dir" are unavailable		
in the API schema as they are unique to the Bloomberg Excel add-ins.		
Sample Code Translation (subscription-based) (click to show/hide)		

BDP Parameters	Description	API Programming Equivalent
Bloomberg Excel Formula (*) (**): BRB("IBM US Equity", "OPEN,HIGH, LAST_PRICE", "6/21/2010 9:30:00AM", "", "BarTp=T", "BarSz=5", "Dir=V", "Dts=S", "Sort=A", "Quote=C", "UseDPDF=Y")		
Equivalent COM v3 Data Control Subscription Code: Dim WithEvents session As blpapicomLib2.Session Dim strOptions As String		
Set session = New blpapicomLib2.session		
session.QueueEvents = True		
session.Start		
session.OpenService "//blp/mktbar"		
Dim sub As blpapicomLib2.SubscriptionList		
Dim cid As blpapicomLib2.CorrelationId		
Set sub = session.CreateSubscriptionList()		
Set cid = session.CreateCorrelationId(1)		
' All times are in GMT		
strOptions = "start_time=14:30&interval=5"		
sub.AddEx "//blp/mktbar/ticker/IBM US Equity", strOptions, cid		
session.Subscribe sub		
* All times in a BRB() intraday calls are in local time,		
whereas in the v3 API they are in GMT. The above BRB()		
time is assumed to be GMT-5:00 (EST)		
** Some parameters in the BDH() function, such as "Dir" are unavailable		
in the API schema as they are unique to the Bloomberg Excel add-ins.		

15.7. BEQS(): BLOOMBERG EQUITY SCREENING

The BEQS() function allows users to retrieve a table of data for a selected equity screen that was created using the Equity Screening (EQS) function.

Service: “//blp/refdata”

Request Type: BeqsRequest

BDP Parameters	Description	API Programming Equivalent
ScreenName	Name of the screen to import. This property is required. For example, “Frontier Market Stocks with 1 billion USD Market Caps” or “MyScreen.” Note: If user is importing a Bloomberg screen, remember to use the ScreenType and Group optional parameters to identify the screen.	Element: screenName Element value: String
ScreenType	Indicates that the screen is a Bloomberg-created sample screen or a saved custom screen that users have created. This property is optional.	Element: screenType Element values (ENUMERATIONS): GLOBAL (B — Bloomberg) or PRIVATE (C — custom*) * default
Group	If the screens are organized into groups, allows users to define the name of the group that contains the screen. This property is optional. If the users use a Bloomberg sample screen, they must use this parameter to specify the name of the folder in which the screen appears. For example, “Group=Investment Banking” (when importing the “Cash/Debt Ratio” screen).	Element: Group Element value: String
AsOf	Allows users to backdate the screen, so they can analyze the historical results on the screen.	Element: overrides Override Name: fieldId (string): PiTDate Override Value: value (string): YYYYMMDD

BDP Parameters	Description	API Programming Equivalent
Lang	<p>Allows users to override the EQS report header language.</p> <p>For a list of valid languages, type LANG <GO> on the Bloomberg Professional service.</p>	<p>Element: languageld</p> <p>Element values (ENUMERATIONS):</p> <p>ENGLISH KANJI FRENCH GERMAN SPANISH PORTUGUESE ITALIAN CHINESE_TRAD KOREAN CHINESE_SIMP THAIK SWED FINNISH DUTCH MALAY RUSSIAN GREEK POLISH DANISH FLEMISH ESTONIAN TURKISH NORWEGIAN LATVIAN LITHUANIAN INDONESIAN</p> <p>* default is the native OS language.</p>
<p>Sample Code Translation (click to show/hide)</p>		

BDP Parameters	Description	API Programming Equivalent
<u>Bloomberg Excel Formula:</u>	<p>BEQS("Fair Value (Level III Assets)","ScreenType=B", "Sec=Y", "Group=General", "AsOf=20140519", "ForceFmt=Y", "ShowSecurityOnly=N", "Lang=1")</p>	<u>Equivalent COM v3 Data Control Request Code:</u>
	<pre>Dim WithEvents session As blpapicomLib2.Session Dim refdataservice As blpapicomLib2.service Dim req As blpapicomLib2.Request Set session = New blpapicomLib2.session session.Start session.OpenService "//blp/refdata" Set refdataservice = session.GetService("//blp/refdata") Set req = refdataservice.CreateRequest("BeqsRequest") req.Set "screenName", "Fair Value (Level III Assets)" req.Set "screenType", "GLOBAL" ' Equivalent to Bloomberg (B) req.Set "Group", "General" req.Set "languageId", "ENGLISH" Dim overrides As Element Set overrides = req.GetElement("overrides") Dim override As Element Set override = overrides.AppendElement() override.SetElement "fieldId", "PiTDate" override.SetElement "value", "20140519" session.SendRequest req</pre>	
	<p>NOTE: Certain parameters in the BEQS() formula are not available via the API and are implemented within the Excel add-in itself; these includes ForceFmt, Sec and ShowSecurityOnly.</p>	

16. Troubleshooting

GATHER ALL NECESSARY INFORMATION TO REPLICATE ISSUE

- Product (B-PIPE, SAPI, DAPI, COM v3 Data Control, etc.)
- BMDS, ASID, SID
- Instruments/fields/date-time range affected
- Type of affected data request (subscription, reference data request, historical data request, etc.)
- Steps to reproduce

USE DEMO TOOL/SAMPLE PROJECT

Replicate the issue by running the Demo Tool or sample project from the SDK package. This can identify issues with the source code by comparing the results from the Demo Tool and whether the issue is universal or only in the local environment.

CONNECTIVITY ISSUES

1. Confirm if host and port are correctly specified.
2. Telnet to host and port and look for server response.
3. For DAPI, confirm if bbcomm.exe is running in task manager.
4. If process is running, run demo tool and see if they can connect.
5. Check for firewall blocking.

DATA DISCREPANCY

1. Check user entitlement.
2. Ensure that the pricing source is consistent between Terminal/Excel/API via {ALLQ <GO>}, {FLDS <GO>}, Demo Tool, etc.
3. Check the source code and make sure that any request parameters that could affect outcome are correctly set up. Request parameters to check include “adjustmentFollowDPDF,” “adjustmentNormal,” “adjustmentSplit,” “pricingOption,” etc.
4. Check the source code and make sure that overrides are correctly requested as follows:

```
Element overrides = request["overrides"];
Element override1 = overrides.AppendElement();
override1.SetElement("fieldId", "BEST_DATA_SOURCE_OVERRIDE");
override1.SetElement("value", "BLI");
```


16.1. TROUBLESHOOTING SCENARIOS

SCENARIO #1: SESSION.START() RETURN FALSE

A Java program is returning error “session.start() return false.” The error is returned after trying to get end-of-day price of U.S. stocks.

1. Check if user is actively logged in to the Bloomberg Professional service on the same machine as application when requesting data using Desktop API.
2. Check if the “bbcomm.exe” process is running in Windows Task Manager-> Process tab when error is displayed.
3. If client notices that bbcomm.exe is not running, run “BBCOMM” command in Start->Run to start the BBComm process and start the Java application.
4. Restart the machine if the following error is returned when client runs “BBCOMM”:

```
2012/11/08 08:34:47.134 [ERROR] INIT: Failed to bind listenPort 8194 errorCode = 10048

2012/11/08 08:34:47.134 [ERROR] INIT: Most probable cause -> another process is
listening on port 8194
```

5. If the problem is resolved, do a clean install of the Bloomberg Professional service and make sure BBCOMM starts without having to manually run BBCOMM.

SCENARIO #2: SPECIFY “SHOW QRM EQUIVALENT” IN DESKTOP API IN C#

Excel spreadsheet displays condition codes for {AAPL UW Equity} by specifying: “=BDH(.....,”CondCodes=S”,.....”.

Programatically:

```
set “includeNonPlottableEvents=true” and “includeConditionCodes=true”:
    request.set(“includeNonPlottableEvents”,true)
    request.set(“includeConditionCodes”,true)
```

Refer to *Services and Schema Reference Guide* for the details of configurable parameters for an Intraday Tick Request.

Note that timestamps are in GMT for API and in local time for Excel (if BDx() formula used).

16.2. BLP API LOGGING

API logging should be enabled only when debugging:

C++/C/PERL/PYHTON — Export following environment variables:

```
BLPAPI_LOGLEVEL=DEBUG
BLPAPI_LOGFILE=c:\blpapilog.txt
```

Java — Add the following parameters when running the Java application:

```
Dcom.bloomberglp.blpapi.logLevel=FINER
Dcom.bloomberglp.blpapi.logfolder=C:\
```

C# — Add listeners by using App.config to the project.
Sample App.config file is as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <switches>
      <add name="BloombergApiTraceSwitch" value="4" />
    </switches>
    <trace autoflush="true" indentsize="2">
      <listeners>
        <add name="FileListener"
            type="System.Diagnostics.TextWriterTraceListener"
            initializeData="C:\\blp\\API\\publishme.log" />
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```