

BLOOMBERG L.P. OPEN API

ENTERPRISE DEVELOPER GUIDE

Version: 1.2
Last Updated: 07/31/2015

Related Documents

Document Name
Core User Guide
Core Developer Guide
Enterprise User 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).

1.	About This Guide.....	6
1.1.	Overview.....	6
2.	Product Coding Differences.....	6
2.1.	The Desktop API and Server API.....	6
2.2.1.	The Desktop API.....	6
2.2.2.	The Server API.....	7
2.2.3.	Desktop API vs. Server API.....	8
3.	Authorization Coding	9
3.1.	Overview.....	9
3.2.	Authorization process.....	10
3.2.1.	General flow.....	10
3.2.2.	Authorization Modes.....	11
3.2.1.	Chaining Applications.....	12
3.2.2.	Generating a token	12
3.2.3.	Example Code – Get Token	12
3.3.	Creating an Identity.....	13
3.3.1.	Example Code: Authorize an Identity using a token	13
3.3.2.	Handling Entitlement Updates	14
3.3.3.	Using an Identity -For N tier architectures:	14
3.3.4.	Maintaining an Identity	15
4.	Server Side User Authorization.....	15
4.1.	Code Example:	15
5.	Failover Coding	16
5.1.	SDK Failover.....	16
6.	B-PIPE Only Services.....	18
6.1.	Market Depth Service (//blp/mktdepth)	18
6.1.1.	Code Examples	19
6.1.2.	Types of Order Books.....	21
A.	Market-by-Order (MBO)	21
B.	Market-by-Level (MBL).....	21
C.	Market Maker Quote (MMQ).....	21
D.	Top Brokers (TOP).....	21

6.1.3.	Order Book Methods.....	22
A.	Replace-By-Position (RBP)	22
B.	Add-Mod-Del (AMD)	22
C.	Replace-by-Broker (RBB)	22
6.1.4.	Subscribing to Market Depth	22
6.1.5.	Handling Multiple Messages (a.k.a. Fragments)	28
A.	Data Response for ADD-MOD-DEL (AMD) Order Books.....	29
B.	MBO-AMD sample subscription output.....	29
C.	Data Response for Request-By-Broker (RBB) Order Books.....	31
D.	MBO-RBB Subscription Output.....	32
E.	Data Response for Request-By-Position (RBP) Order Books	34
F.	MBL-RBP Subscription Output	35
G.	Order Book Recaps	37
H.	Gap Detection.....	37
6.2.	Market List Service (//blp/mktlist).....	38
6.2.1.	Code Examples	40
6.2.2.	Subscribing To Instrument Chains	40
6.2.3.	Chain Subservice Examples	41
6.2.4.	List Actions	44
6.2.5.	Data Response For a "chain" Subscription	44
6.2.6.	Handling Multiple Messages (a.k.a. Fragments)	46
6.2.7.	Snapshot Request for List of Security Identifiers	46
6.2.8.	Data Response For "secids" Snapshot Request.....	48
6.3.	Source Reference Service (//blp/srcref)	52
6.3.1.	Overview.....	52
6.3.2.	Important BPOD Upgrade Notes:	53
6.3.3.	Code Example	53
6.3.4.	Response Overview.....	54
6.3.5.	Response Event Types by Subservice	54
6.3.6.	Breakdown of Event Type Fields.....	55
6.3.7.	Handling Multiple Messages (a.k.a. Fragments)	56
6.3.8.	Data Response for Subscription	57
6.4.	Message Scraping Service (//blp/msgscrape)	60

6.4.1.	Introduction	60
6.4.2.	Heartbeat:.....	61
7.	New Book Discovery Service	62
7.1.	Requirements	62
7.1.1.	Introduction	62
7.1.2.	Request	62
7.1.3.	Response	63
7.1.4.	System configuration	64
7.1.5.	Performance metrics.....	64
7.1.6.	Service schema.....	64

1. About This Guide

The Enterprise Developer's guide will form the basis for understanding the concepts of different products like **Desktop API**, **Server API**, **B-PIPE** and **Platform** products. In addition users will learn about Authorization and Failover concepts.

This guide is the starting point for learning the core usage of the Bloomberg API libraries. This knowledge will form the basis for developing applications for the Desktop API, Server API, B-PIPE and Platform products.


1.1. Overview

All the 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.

2. Product Coding Differences

2.1. The Desktop API and Server API

The Desktop API and Server API have the same programming interface and behave almost identically. The chief difference is that customer applications using Server API have some additional responsibilities. Those additional requirements will be detailed later in this document (see Enterprise User Guide for Authorization and Permissioning); otherwise, assume the two deployments are identical.

 *Note that in both deployments, the end-user application and the customer's active BLOOMBERG PROFESSIONAL service share the same display/monitor(s).*

2.2.1. The Desktop API

The Desktop API is used when the end-user application resides on the same machine as the installed BLOOMBERG PROFESSIONAL service and connects to the local Bloomberg Communications Server (**BBComm**) to obtain data from the Bloomberg Data Center (see Figure below).

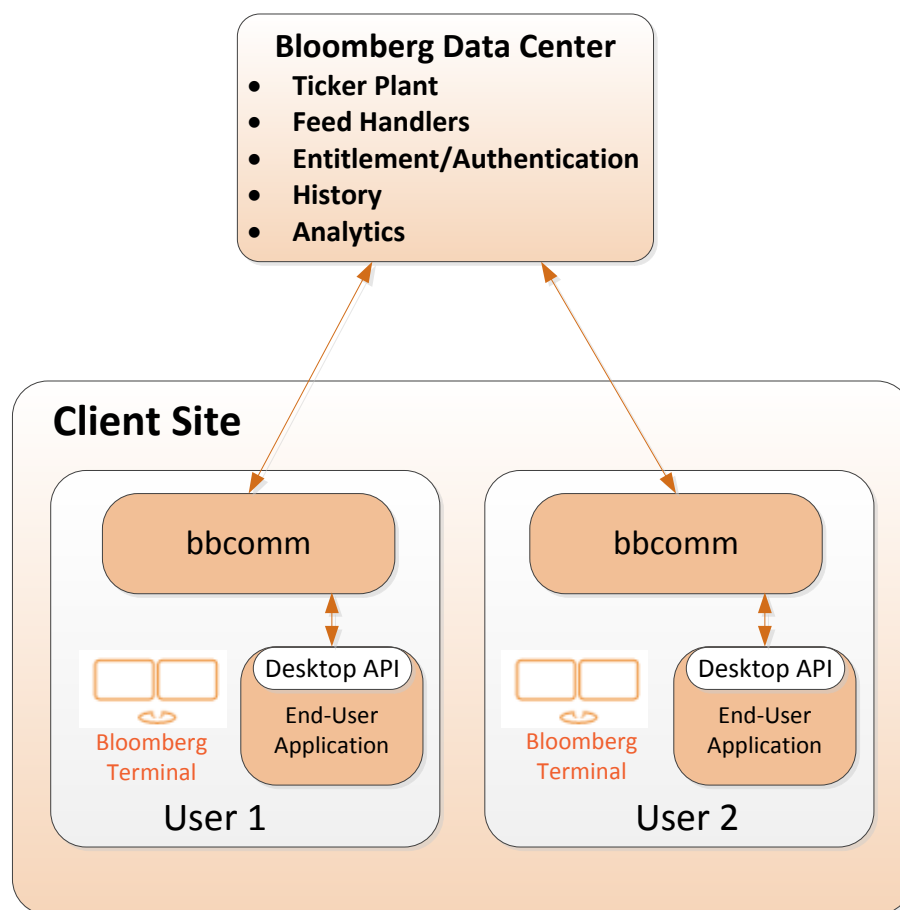


Figure 1: Desktop API

2.2.2. The Server API

The Server API allows customer end-user applications to obtain data from the Bloomberg Data Center via a dedicated process, known as the Server API process. Introduction of the Server API process allows, in some circumstances, better use of network resources.

When the end-user applications interact directly with the Server API process they are using the Server API in User Mode.

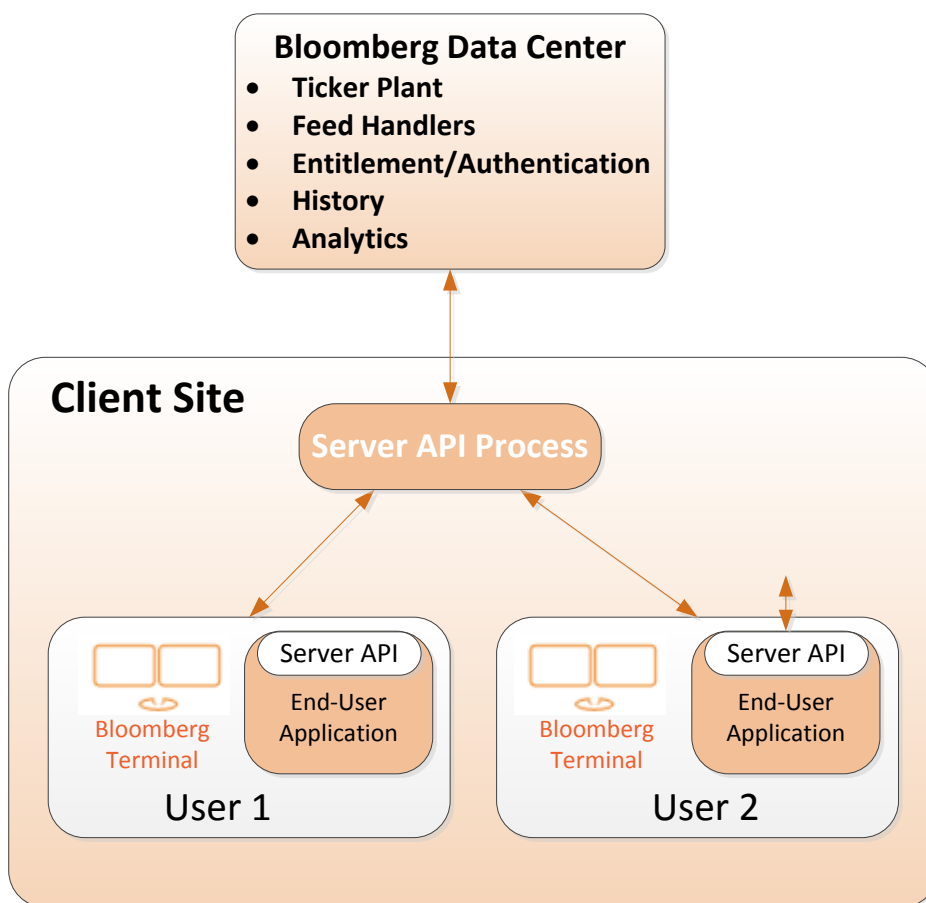


Figure 2: Server API – User Mode

2.2.3. Desktop API vs. Server API

Both the Desktop API and Server API use the identical API libraries and do not allow non-BPS users to retrieve data. The chief programming difference is that customer applications using the Server API have some additional responsibilities such as authorization and permissioning before distributing/receiving data.

Note: In both deployments, the end-user application and the customer's active Bloomberg Professional service must share the same display/monitor(s).

3. Authorization Coding

3.1. Overview

The table below summarizes the key differences between different types of authorization:

	Desktop API user mode	Server API User Identity: UUID/IP	Enterprise : AUTHID/IP	Enterprise : Token	Comments
Identity required	No Best practice: Create an identity with no auth step performed:	Yes	Yes	Yes	IP must be from the PC where <ul style="list-style-type: none"> - If BBG, where the user is logged into the BBG Terminal - the application is being displayed The IP must be dynamically obtained
Auth using:	Skip Auth	UUID/IP	AUTHID/IP	Token	Set in <code>SessionOption's AuthenticationOptions</code>
Notes	Using an identity will make porting to other API products easier	No entry in SAPE is required:		<code>GenerateToken()</code> will obtain credentials based on auth options and will dynamically obtain the IP where it is run	Enterprise mode requires users and applications are in SAPE<GO> for Server API and otherwise EMRS<GO>
Code	<code>id=create();</code> <code>If (!Desktop)</code> <code>Auth(id)</code>	<code>authRequest.set("uuid", UserUUID);</code> <code>authRequest.set("ipAddress", ip);</code>	<code>authRequest.set("AUTHID", UserUUID);</code> <code>authRequest.set("ipAddress", ip);</code>	<code>String UTok =getToken();</code> <code>authRequest.set("token", UTok);</code>	Identities are created from a session, and can only subscribe or request data from that session Permission calls may be made across sessions

3.2. Authorization process

3.2.1. General flow

Displayed below is a figure that displays the Authorization process and the steps involved:

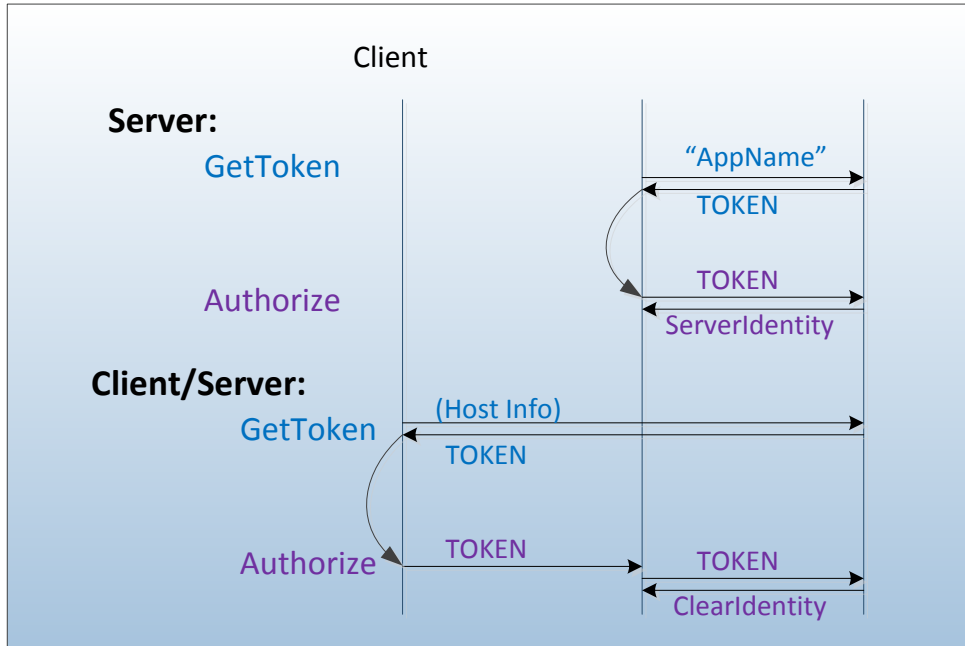


Figure 3: Authorization Process Flow

1. The Server creates a **Session** with the Bloomberg Servers and then gets an **application token** for itself.
2. The Server then authorizes the token and can now use **ServerIdentity** for subscriptions and requests. Users do not have to be logged in for this to take place. The server **MUST** keep all EIDs associated with any data received and associate these EIDs with any calculations made on the data.
3. The Client now creates a Session with Bloomberg Servers.
4. Next, the client generates a token for itself and sends it to the server.
5. Client now closes the session.
6. The Server creates an identity using the token in the **Authorization** call.
7. Before passing any data to a specific client, the Server **MUST** assert to make sure that the client's identity is valid and also checks for all associated EIDs for permission using client's identity.

Note that, while subscriptions and requests can only be used with the Session that is used to create the identity, the identity may also be used with data from any session for permissioning purposes. The session used to create the client identity however must be kept active for events related to that identity to propagate.

8. The server **MUST** cancel, not just delete an identity to release the seat and total the active connection count. If the identity has been revoked, it **MUST** be deleted NOT cancelled. In case of NONBPS clients, for products that support it, the server may cancel an identity if the user wants to

switch to a new IP. The NONBPS user is not allowed to have an identity created for a new IP, unless all servers have canceled their identity associated with the old IP. NONBPS users with seat counts greater than 1 will have the same constraint when trying to exceed their seat count.

3.2.2. Authorization Modes

AuthorizationOptions: This is a Member variable of SessionOptions and should be set to one of the following:

Operating System Login (Domain\User):


`"AuthenticationType=OS_LOGON"`

This is the login that the application will run as; generally this is the login at the display device. Since many operating systems allow multiple concurrent logins, this is not sufficient to uniquely identify a user and the IP will either implicitly through getToken, or explicitly through server side authorization be used. It is never the Server IP, unless the user is displaying back to the IP where the Server API is running. This is however not recommended.

Active Directory:

`"AuthenticationType=DIRECTORY_SERVICE; DirSvcPropertyName=<ENTRY>"`

The "<" and ">" should NOT be entered. This is only available on Windows, and entries vary from firm to firm. The Active Directory entry used MUST be unique per user. Employee ID and Email are good examples. Firm name, location or department are not unique per user and may not be used.

 *Note that the Active Directory entry name should be used as ENTRY and not the value, for example:*

Use `"...; DirSvcPropertyName=EmpID"` and not `"...; DirSvcPropertyName=161238756"`

Email (User@firm.com):

`"AuthenticationType=DIRECTORY_SERVICE; DirSvcPropertyName=mail"`

Generally email is stored in Active Directory in the "mail" property. This is just a common use case for Active Directory. Do not use the users actual email address.

Application Mode:

`"AuthenticationMode=APPLICATION_ONLY;
ApplicationAuthenticationType=APPNAME_AND_KEY; ApplicationName=<APP
NAME>"`

The "<" and ">" should not be entered. This is the only way to authorize an application as entered into EMRS<GO> or SAPE<GO> as an application. Applications are not users; they are a different entry type. EMRS<GO> allows limiting applications to specific IPs or ranges. Servers using the same application name may run multiple instances on multiple hosts. There is a reasonably high limit to the number of concurrent usage. There is a possibility the associated identity can be revoked, so coding should be in place to reauthorize the identity in response to the revoke event.

USER+Application Mode (OS_LOGON):

`"AuthenticationMode=USER_AND_APPLICATION;`

```
ApplicationAuthenticationType=APPNAME_AND_KEY;  
ApplicationName=<APP NAME>; AuthenticationType=OS_LOGON"
```

This will enforce that the user has been added with the OS_LOGIN user id or alias and has been added to that application in **EMRS**<GO> or **SAPE**<GO>. It will also use the union of EID permissions between the user and the application. NONBPS users cannot be added to BPS applications. See "[Operating System Login](#)" above for additional information.

USER+Application Mode (Active Directory/email):

```
"AuthenticationMode=USER_AND_APPLICATION;  
ApplicationAuthenticationType=APPNAME_AND_KEY;  
ApplicationName=<APP NAME>;  
AuthenticationType=DIRECTORY_SERVICE; DirSvcPropertyName=<ENTRY>"
```

This will enforce that the user has been added with the corresponding ENTRY value as the user id or alias and has been added to that application in **EMRS**<GO> or **SAPE**<GO>. It will also use the union of EID permissions between the user and the application. NONBPS users cannot be added to BPS applications. See "[Active Directory](#)" above for more detail.

3.2.1. Chaining Applications

API supports in having an application as a client of another application and this may in turn have users and applications as its client. This is currently not reflected in **EMRS**<GO> or **SAPE**<GO>. To process requests from an application; the client application will open a session, create a token, and then close that session. Once it sends the token to the server application, the server application can treat the same like any other token for permissioning purposes, including checks for EID permissions and validating BPS or NONBPS seat types.

3.2.2. Generating a token

Having defined AuthorizationOptions and started a session, call the GenerateToken method of the session object. Use an event loop to poll for a TOKEN_STATUS event.

TOKEN_STATUS: This is the status update for generating token events.

MessageType: "TokenGenerationSuccess" Msg.getElementAsString("token") will yield token "TokenGenerationFailure". Msg contains reason.

3.2.3. Example Code – Get Token

```
C++  
Name TokenGenerationSuccess("TokenGenerationSuccess");  
Name TokenGenerationFailure("TokenGenerationFailure");  
string demo::getToken()  
{  
    EventQueue eQ;  
    session_ ->generateToken(nextId(), &eQ);
```

```

while (1)
{
    Event event=eQ.nextEvent();

    if (event.eventType() == Event::TOKEN_STATUS)
    {
        MessageIterator iter(event);
        while(iter.next())
        {
            Message msg=iter.message();
            if (msg.messageType() == TokenGenerationSuccess)
                return msg.getElementAsString(Token);
            if (msg.messageType() == TokenGenerationFailure)
                return "";
        }
    }
}
}

```

3.3. Creating an Identity

The steps to create an Identity are as follows:

1. Call the session's **CreateIdentity** method.
2. Open the **//blp/apiauth** service and use it to create an **AuthorizationRequest**.
3. Populate the **AuthorizationRequest** with the **UUID** and **IP**.
4. Send the request along with the newly created Identity via the session's **SendAuthorizationRequest** method.
5. Next, monitor for **PARTIAL_RESPONSE** events with an AuthorizationSuccess message.
PARTIAL_RESPONSE (RESPONSE for Failure events)
 User authorization status event
 MessageType:
 "AuthorizationSuccess"
 Identity is correctly instantiated with permissions
 "AuthorizationFailure"
 Identity is invalid – token provided is likely invalid

3.3.1. Example Code: Authorize an Identity using a token

Use the following to associate the token with an identity

```
authRequest.set("token", token);
```

In context it is used as follows to authorize an identity:

```

C++
const char * AUTH_SVC="//blp/apiauth";
Name AuthorizationSuccess("AuthorizationSuccess");
int d_uuid = "12345";
int ipAddress = "127.0.0.1";

Identity demo::getIdentity(string tok)
{

```

```

Identity ID=session_>createIdentity();
EventQueue eQ;
Service auth = session_>getService(AUTH_SVC);
Request authRequest = auth.createAuthorizationRequest();
authRequest.set("token", tok);
session_>sendAuthorizationRequest(authRequest, &ID, nextId(), &eQ);
while(1)
{
    Event event=eQ.nextEvent();

    if (event.eventType() == Event::RESPONSE
        || event.eventType() == Event::PARTIAL_RESPONSE
        || event.eventType() == Event::REQUEST_STATUS)
    {
        MessageIterator iter(event);
        while(iter.next())
        {
            Message msg=iter.message();
            if (msg.messageType() == AuthorizationSuccess)
                return ID; // Success
        }
        return ID; // Failure: ID is invalid
    }
}

```

3.3.2. Handling Entitlement Updates

Once the Identity is successfully created, changes may occur to the permissions it represents. If a Bloomberg user logs into a new host, a **REVOKE** event will occur. The specified eventQueue should not be used; however the prior examples used an eventQueue for simplicity.

The REVOKE and UPDATE events must be handled, if not they will be lost if the eventQueue goes out of scope. All authorization related events need to return via an event handler or the default queue and keep the eventQueue active in a thread and continue monitoring. As a result, the Identity management system in an application needs to be asynchronous. All events referring to a specific authorization request will be of the type PARTIAL_RESPONSE. A RESPONSE event type will only occur on revocation / termination of an Identity and thus its authorization request.

3.3.3. Using an Identity -For N tier architectures:

In this case the Server Identity is added to all subscriptions and requests, for example:

```

Session.subscribe(subscriptionList, ServerIdentity);
Session.sendRequest(request, ServerIdentity, correlationId, eventQueue);

```

The Client Identity is used to check permissions and seat type as shown below:

```

ClientIdentity.hasEntitlements(neededEntitlements, message.service(),
missingEntitlements);
Identity::SeatType seatType=ClientIdentity.getSeatType();

```

Where the **seatType** can be:

BPS – Valid Bloomberg User

NONBPS – Valid Market Data User (Non Bloomberg Terminal User)

INVALID_SEAT – Not a valid user

Once the Identity is generated there is no need to generate the token. In case of **Client Only/Black Box/Derived Data Apps**, generate the Identity locally, and use it for subscriptions/requests. Clients will use **OS_LOGIN** or **email**, Black Box/Derived Data Apps will use their **EMRS application name**.

3.3.4. Maintaining an Identity

This is a monitor for AUTHORIZATION_STATUS events. In case of **AUTHORIZATION_STATUS**, this is a User authorization status update event where the

MessageType: "AuthorizationRevoked" (Update to Identity – Authorization no longer granted) and "EntitlementChanged" (Update to Identity – Permissions updated).

4. Server Side User Authorization

All Server API products support UUID and IP as credentials that an application can use to create a user identity instead of using a token generated on the user's desktop. All products support using AUTHID and IP as credentials that an application can use to create a user identity.

In a client/server application the server needs to permission its clients but there are cases where the client will not be able to create a token. To list a few of them:

- Client has no connectivity to B-PIPE
- Remote display access – the IP is not always the display IP – the API will handle most Citrix use cases
- Web Applications – client is a browser so code cannot easily be added

For permissioning in these cases a client identity can be generated in the server from any EMRS or SAPE name or alias and the Display IP:

- No token is used
- The Identity is authorized using AUTH ID2, and IP instead of Token
- All other coding is the same
- Note that all IPs must be dynamically, and programmatically determined and not stored in config files or the like

4.1. Code Example:

The following code snippet is used to authorize an identity using a token:

```
authRequest.set("token", token);
```

It is replaced in server side authorization as follows:

Server API Only:

```
authRequest.set("uuid", UserUUID);
authRequest.set("ipAddress", ip);
```

All products:

```
authRequest.set("AUTHID", UserEntry);
authRequest.set("ipAddress", ip);
```

 Note that both entries must be used.

UserEntry must be the same as entered in **EMRS<GO>** or **SAPE<GO>**. It may be a user name or alias but cannot currently be an application. The “**AUTHID**” label replaces the old “**emrslid**” label, in order to be inclusive for **SAPE<GO>** support, all functionality remains the same, and currently either label will work.

5. Failover Coding

5.1. SDK Failover

The SDK supports Live-Backup failover in the event of network failure between the application and a Bloomberg communication server (appliance). To enable this feature, multiple server host IPs must be provide to the `SessionOptions.setServerAddress` and set `SessionOptions.setAutoRestartOnDisconnection` to true.

To enable failover to occur within the API, `sessionOptions.setAutoRestartOnDisconnection` must be set to TRUE.

The number of retries is by default set to one. And will occur every 2 seconds. This can be adjusted by setting `SessionOptions.setNumStartAttempts` to the number of asynchronous attempts to reconnect.

In general this can be set to a really high number. The code example on setting up failover is as displayed:

```
<C++>
private string firstIP = "10.1.2.3";
private string secondIP = "10.1.2.4";
private SessionOptions sessionOptions;
private Session *session;

sessionOptions.setServerAddress(firstIP.c_str(), 8194, 0);
sessionOptions.setServerAddress(secondIP.c_str(), 8194, 1);
sessionOptions.setAutoRestartOnDisconnection(true);

// setting the number of start attempts to a high number
sessionOptions.setNumStartAttempts(1000000);

// other SessionOptions setting here

// start session
session = new Session(sessionOption, eventHandler);
session->start();
```

SDK Failover: Connection Issues Scenarios

Here are some common connection issues that may occur when performing failover in the application and the following steps to resolve them.

A. Both host IP and port pairings are unreachable

1. Call `Session.start()`.
2. The API will internally attempt to connect to all provided IP and port pairings.
3. Obtain a `SESSION_STATUS` event with a *SessionStartupFailure* message when all connection attempts have been exhausted.

B. Connect and disconnect to first IP and port then failover to second IP and port

1. Call `Session.start()`.
 - a. Get `SESSION_STATUS` event with *SessionConnectionUp* message for the first IP and port.
 - b. Get `SESSION_STATUS` event with *SessionStarted* message.
2. Network issue between first host IP and application.
 - a. Get `SESSION_STATUS` event with *SessionConnectionDown* message for the first IP and port.
 - b. API tries to connect to second IP and port.
 - c. Success connection to second IP and port.
 - o Get `SESSION_STATUS` event with *SessionConnectionUp* message for the second IP and port.
 - o Re-subscribe to excising subscriptions.
 - o Any outstanding reference data requests are lost and require resend of requests.
 - o Retry count rest.
3. Network issue between second host IP and application.
 - a. Get `SESSION_STATUS` event with *SessionConnectionDown* message for the second IP and port.
 - b. API will try to connect to first IP and port.
 - c. Success connection to first IP and port
 - I. Get `SESSION_STATUS` event with *SessionConnectionUp* message for the second IP only.
 - II. Re-subscribe to excising subscriptions.
 - III. Any outstanding reference data requests are lost and require resend of requests.
 - IV. Retry count rest.
4. Connect and disconnect to first IP and port then fail to connect to all host IPs
 - a. Call `Session.start()`.
 - I. Get `SESSION_STATUS` event with *SessionConnectionUp* message for the first IP and port.
 - II. Get `SESSION_STATUS` event with *SessionStarted* message.

C. Network issue between first host IP and application.

1. Get `SESSION_STATUS` event with *SessionConnectionDown* message for the first IP and port.
2. API tries to connect to second IP and port.

3. Failed to connect to second IP and port, try to connect to first IP and port. No event generated since there was no connection. (start attempt 1)
4. Failed to connect to first IP and port, try to connect to second IP and port. No event generated since there was no connection. (start attempt 1)
5. Failed to connect to second IP and port, try to connect to first IP and port. No event generated since there was no connection. (start attempt 2)
6. Failed to connect to first IP and port, try to connect to second IP and port. No event generated since there was no connection. (start attempt 2)
7. Get `SESSION_STATUS` event with *SessionTerminated* message, as the API has reached the number of tries set with the `SessionOptions.numStartAttempts` property.

6. B-PIPE Only Services

This section will expand upon each of the four services specific to only B-PIPE developers.

B-PIPE provides access to the full 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 and thus a different EID is typically used for "level 2". The services are as follows:

- **Market Depth Service (`//blp/mktdepthdata`)**
- **Market List Service (`//blp/mktlist`)**
- **Source Reference Service (`//blp/srcref`)**
- **Message Scraping Service (`//blp/msgscrape`)**

Field filtering is available as a configuration option, which means that B-PIPE clients have the option to change their configurations so that only the fields specified in a subscription are returned. As a result, clients should be able to recognize significant bandwidth savings on their Client LAN.

6.1. Market Depth Service (`//blp/mktdepth`)

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 full 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 and thus a different EID is typically used for "level 2". Market depth, order books and level 2 data are all names for the same set of data. They provide information about the bid and ask prices that currently exist for an instrument.

Generally, the "top of the book", i.e., the price in the top row (row 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. If this occurs then it is known as a crossed or inverted market (or book). The details of the specific conditions vary by market.

Many times exchanges consider order book (level 2) information a separate product from its level 1 data and charge additional fees for access to it. In these cases the level 2 data will have a different EID than the level 1 data. Order books have three characteristics that define them: The number of rows in the book (window size), the type of the order book and the method used to update the book.

There are 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. There are three order/quote book update methods, **Replace-By-Position** (RBP), **Add-Mod-Delete** (AMD) and **Replace-By-Broker** (RBB).

The Market Depth service is subscription-based and allows the subscription to all levels of market depth data. It is available to both BPS (Bloomberg Professional Service) and Non- BPS users.

Before delving into the market depth service and its data, let's first take a look at another way to obtain limited market depth data via the already existing `//blp/mktdata` service. With this service, one can obtain up to the first 10 levels of market depth by level (aka MBL) data. This is accomplished by making a `//blp/mktdata` subscription and including one or more of the following fields:

Mnemonic	Description
BEST_BID1 thru BEST_BID10	First thru tenth best bid price in ten levels of market depth
BEST_BID1_SZ thru BEST_BID10_SZ	Size of first thru tenth best bid in ten levels of market depth
BEST_ASK1 thru BEST_ASK10	First thru tenth best ask price in ten levels of market depth
BEST_ASK1_SZ thru BEST_ASK10_SZ	Size of first thru tenth best ask in ten levels of market depth

Keep in mind that this method of obtaining market depth through the `//blp/mktdata` service is limited to receiving only aggregated Market By Level data for up to 10 levels. This service doesn't allow users to obtain "Market By Order" (MBO) data. Also, the `//blp/mktdata` service does not provide them with information such as the book type or the action performed on that position.

Therefore, if users wish to receive more than 10 levels of market depth by level (MBL) or any market depth by order (MBO) levels, then they will be required to use the `//blp/mktdepthdata` service. Subscribing to this comprehensive service will not only supply them with the order book in its entirety, but also provide the book type, action performed, etc.

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

6.1.1. Code Examples

In this section will find two separate examples in the B-PIPE SDK for C++, Java and .NET. They are as follows:

- **MarketDepthSubscriptionExample**

This example demonstrates how to make a simple Market Depth subscription for one, or more, securities and display all of the messages to `std::cout`. A snapshot of this example is displayed below:

```

// m_service: the service name. Set to "//blp/mktdepth" in this case.
// m_fields: fields array. Typically empty for mktdepth.
// m_topics: the topic to subscribe to, "/ticker/IBM US Equity", for example.
// m_options: the service request options. Examples are "type=MBO", "view=LIMITED".
// m_hosts: array of host names. One for each port.
// m_ports: array of ports. One for each host.
// m_authOptions. String for authentication. Empty for no authentication.
void subscribe()
{
    SessionOptions sessionOptions;
    for (size_t i = 0; i < m_hosts.size(); ++i)
    {
        sessionOptions.setServerAddress(m_hosts[i].c_str(), m_ports[i], i);
    }
    sessionOptions.setAuthenticationOptions(m_authOptions.c_str());
    sessionOptions.setAutoRestartOnDisconnection(true);

    // NOTE: If running without a backup server, make many attempts to
    // connect/reconnect to give that host a chance to come back up (the
    // larger the number, the longer it will take for SessionStartupFailure
    // to come on startup, or SessionTerminated due to inability to fail
    // over). We don't have to do that in a redundant configuration - it's
    // expected at least one server is up and reachable at any given time,
    // so only try to connect to each server once.
    sessionOptions.setNumStartAttempts(m_hosts.size() > 1 ? 1 : 1000);

    Session session(sessionOptions);
    if (!session.start())
    {
        std::cerr << "Failed to start session." << std::endl;
        return;
    }

    Identity subscriptionIdentity(session.createIdentity());
    // ...
    // Authenticate here, if required.
    // ...
    SubscriptionList subscriptions;
    for (size_t i = 0; i < m_topics.size(); ++i)
    {
        std::string topic(m_service + m_topics[i]);
        subscriptions.add(
            topic.c_str(),
            m_fields,
            m_options,
            correlationId((char*)m_topics[i].c_str()));
    }
    session.subscribe(subscriptions, subscriptionIdentity);

    return;
}

```

Figure 4: MarketDepthSubscriptionExample

- MarketDepthSubscriptionSnapshotExample**

This example demonstrates how to build and update an order and level book. It is comprised of a LevelBook and OrderBook class, which handle the Market Depth By Level and By Order messages, respectively, based upon the returned **MD_TABLE_CMD_RT** value, and then the main classes which perform the subscription, general message handling and output tasks.



Figure 5: MarketDepthSubscriptionSnapshotExample

Number of Rows in an Order Book

The number of rows in a book may be limited or not. Many exchanges limit their books to as few as 5 rows (positions), others may have as many as 200 rows while still others may not have a predefined limit to the number of rows a book may have. The number of rows that are sent to a client can also be limited by the vendor providing the data. In general, 200 rows are considered a large book. When an order book has a

limited size, and most do, prices or orders can be dropped and added back regularly as the top of the book changes. There is no connection between the number of rows in a book and the type and method of the book. Each is independently determined by the source of the book.

6.1.2. Types of Order Books

There are three types of book; **Market-By-Order** (MBO), **Market-By-Level** (MBL) and **Market Maker Quote** (MMQ). An exchange operating an order book could provide MBO only, MBL only, or both. In some cases, the exchange provides an MBO book and the MBL book is derived by Bloomberg. It is possible to aggregate an MBO into an MBL book, but it is not possible to split an MBL book into its component orders. An exchange operating a quote book would provide MMQ. In some rare instances, a given security may support both an order book (MBO and/or MBL) and a quote book (MMQ), if the market supports both trading mechanisms on the same security. An example of such a market is the SETSxq market at London Stock Exchange.

A. Market-by-Order (MBO)

An MBO book provides every order in the book, subject to the constraints defined by the view and window size attributes. If multiple brokers have orders at the same price level the book will show each order, resulting in multiple rows sharing the same price. The amount of data that is available at each level varies by the source of the data but it typically consists of the price, size, time of the order and in some instances a broker ID. Positions are amended or removed from MBO books as orders are matched and partially or completely executed on the exchange.

B. Market-by-Level (MBL)

An MBL order book is the aggregated market-by-price/yield (previously often called Market-By-Level). This displays only one position (row) for each unique price. If multiple brokers have the same price, then the size of all of their orders will be accumulated and displayed against that price.

As orders are matched and executed at the exchange, the volume available at a price may be completely or partially consumed and updates are provided so that clients can represent the available price and volume as market conditions change.

C. Market Maker Quote (MMQ)

An MMQ book provides a collection of all of the competing quotes from each of the brokers or market makers on a security. There are usually only two quotes (one best bid offer quote, and one best ask offer quote) from each participant, commonly referred to as two-way quotes, and these represent the prices at which that participant is obliged to buy or sell during a mandatory quotation period (hence they “make the market”). All participants compete against one another, and it is possible to rank the quotes in the MMQ book and thus build a virtual aggregated price book.

D. Top Brokers (TOP)

Top brokers is primarily Used for Hong Kong Exchange (HKEx) to provide the top 40 broker orders on each side of the market, but with prices only (no volumes).

6.1.3. Order Book Methods

A. Replace-By-Position (RBP)

In replace-by-position book management, a specific set of columns (size, number of orders, time, etc.) varies by exchange. Often, an update to one level in an RBP book will cause changes to other levels in the RBP book. When many levels are updated as part of the same event, the Multi-tick Update (MTU) flag may be included (if the MTU attribute value is equal to "ON") so that clients will know when all updates are complete and the book is returned to a valid state.

This approach can be used for both MBO and MBL types of books. The updated methodology is straightforward. Clients should just locate the position that is specified and overwrite the price, size and time at that position with the new data that has been supplied.

B. Add-Mod-Del (AMD)

The second order book method is Add-Mod-Delete (AMD). It is used for both MBO and MBL types of order books. The AMD method is much more efficient in sending updates to order books. Instead of addressing each row in the book individually only the changes to the book are sent. This means that client applications must manage any related updates resulting from an Add or Delete event.

For instance, when a new price is inserted at a specific row, the only message sent is the insert. It is the application's responsibility to adjust the position of all the rows that have been shifted down. Likewise, when a row is deleted, it is the application's responsibility to shift all the prices that were below it up. Of course any new price at the bottom of the book requires a separate "Insert", but this is much more efficient than resending the whole book.

Because a single AMD message can affect a single row, one missed message can result in the order book being wrong for the rest of the day or until a recap is sent. Because of this, AMD messages are sent using sequence numbers. If the application detects a gap in the sequence numbers it can recover from the error by re-requesting the entire order book. In other words, resubscribe to the book. If the gap is detected as a result of an issue within the Bloomberg Data Center, Bloomberg will send down an order recap. This form of gap detection is covered in a later section.

C. Replace-by-Broker (RBB)

In replace-by-broker books, the bid/ask for a specific broker is communicated as a replacement of the bid/ask data that had previously been held for that broker.

This style is used solely for MMQ book types and it is a mixture of RBP and AMD update types; the book is built from broker entries and it is similar to the RBP message in that rows are directly indexed (by row in RBP and by broker code in RBB).

How RBB order books are sorted is left up to the consuming application. The general rule is to follow price > time > size priority.

6.1.4. Subscribing to Market Depth

The first step in subscribing to the `//blp/mktdepthdata` service is to learn how the subscription strings are formulated. For the string to be valid, users must specify a "type" parameter, which can be either MBO

(Market by Order) or MBL (Market by Level). Users cannot specify more than one of these in a subscription string. This is appended to the end of the string, immediately following the "?" delimiter.

Here is a list of valid market depth subscription string formats, along with an example of each.

Key Field	Format	Example
Ticker	//blp/mktdepthdata/ticker/symbol	//blp/mktdepthdata/ ticker /VOD LN Equity?type= MBO
ISIN	//blp/mktdepthdata/isin/ identifier source	//blp/mktdepthdata/ isin /DE0005557508 TQ?type= MBL
CUSIP	//blp/mktdepthdata/cusip/ identifier source	//blp/mktdepthdata/ cusip /459200101 LN?type= MBL
SEDOL	//blp/mktdepthdata/sedol/ identifier source	//blp/mktdepthdata/ sedol /0540528 TQ?type= MBL
Bloomberg Unique ID	//blp/mktdepthdata/buid/ identifier source	//blp/mktdepthdata/ buid /EQ000000000496862 JT?type= MBL
BSID	//blp/mktdepthdata/bsid/ bsid	//blp/mktdepthdata/ bsid /2005750482138?type= MBL
ID_BB_Global	//blp/mktdepthdata/bbgid/ bbgid /bbgid source	//blp/mktdepthdata/ bbgid /BBG000BDQGR5 IX?type= MBL
CATS	//blp/mktdepthdata/cats/ identifier source	//blp/mktdepthdata/ cats /6888 MK?type= MBL
CINS	//blp/mktdepthdata/cins/ identifier source	//blp/mktdepthdata/ cins /G0408V102 US?type= MBO
COMMON	//blp/mktdepthdata/common/ identifier source	//blp/mktdepthdata/ common /025929551 LN?type= MBO
SICOVAM	//blp/mktdepthdata/sicovam/ identifier source	//blp/mktdepthdata/ sicovam /013000 FP?type= MBL
SVM	//blp/mktdepthdata/svm/ identifier source	//blp/mktdepthdata/ svm /356573 BB?type= MBL
WERTPAPIER	//blp/mktdepthdata/wpk/ identifier source	//blp/mktdepthdata/ wpk /803200 GY?type= MBL
AUSTRIA	//blp/mktdepthdata/austria/ identifier source	//blp/mktdepthdata/ AUSTRIA /080905 AV?type= MBL
BELG	//blp/mktdepthdata/belg/ identifier source	//blp/mktdepthdata/ BELG /381027 BB?type= MBL
Bloomberg Symbol	//blp/mktdepthdata/bsym/ source/symbol	//blp/mktdepthdata/ bsym /LN/VOD?type= MBL //blp/mktdepthdata/ bsym /US/AAPL?type= MBO
Parsekeyable	//blp/mktdepthdata/bpkbl/ bpkbl	//blp/mktdepthdata/ bpkbl /QCZ1 Index?type= MBL
FRENCH	//blp/mktdepthdata/french/ identifier source	//blp/mktdepthdata/ french /013000 FP?type= MBL
IRISH	//blp/mktdepthdata/irish/ identifier source	//blp/mktdepthdata/ IRISH /3070732 ID?type= MBL
VALOREN	//blp/mktdepthdata/valoren/ identifier source	//blp/mktdepthdata/ VALOREN /002489948 VX?type= MBL

The following C++ code snippet demonstrates how to subscribe for streaming (MBL) market depth data and assumes that a session already exists and that the "//blp/mktdepthdata" service has been successfully opened.

```
const char *security =
    "//blp/mktdepthdata/isin/US/US4592001014?type=MBL"; SubscriptionList
    subscriptions;

    subscriptions.add(security, CorrelationId((char
    *)security)); session.subscribe (subscriptions);
```

Response Overview

The Market Depth response will be a series of SUBSCRIPTION_DATA events, which users will already be familiar with if they have developed Bloomberg API applications using any of the other streaming services, such as //blp/mktdata or //blp/mktvwap.

A SUBSCRIPTION_DATA event message will be of type MarketDepthUpdates, and within each message there will be a MKTDEPTH_EVENT_TYPE and MKTDEPTH_EVENT_SUBTYPE field, along with, possibly, an array of MBO_TABLE_ASK/ MBO_TABLE_BID items (for MBO subscription) or MBL_TABLE_ASK/MBL_TABLE_BID (for MBL subscriptions).

The MKTDEPTH_EVENT_TYPE will indicate whether the message is Market by Level (value= MARKET_BY_LEVEL) or Market by Order (value = MARKET_BY_ORDER). Here are the possible values for each MKTDEPTH_EVENT_SUBTYPE:

MKTDEPTH_EVENT_SUBTYP	Notes
TABLE_INITPAINT	<p>This is the Initial Paint message for the subscription</p> <p>When this message is received, it is an indicator to the user to clear the book cache and add the rows contained in the message.</p> <p>This message will contain the FEED_SOURCE, ID_BB_SEC_NUM_SRC (a.k.a. BSID) and MD_BOOK_TYPE. No other messages will contain this information, so it is required that the user should assign a unique correlation identifier to each one of their subscriptions in order to map the message updates to the initial request.</p> <p>For AMD and RBP book types, there will be a WINDOW_SIZE field/ value pairing, which indicates the number of levels in the book, as position is the key to the book. However, this field will not be contained in the MBO-RBB initial paint, as the key for this book is the broker.</p>
BID	This indicates a bid quote message
ASK	This indicates an ask quote message
BID_RETRANS	In the event of a loss of connectivity upstream, the Bloomberg infrastructure will automatically recover (RECAP) and send BID_RETRANS and ASK_RETRANS events. Upon receipt of these messages, user will receive a CLEARALL message with a MKTDEPTH_EVENT_SUBTYPE of RETRANS and they should consider their book in a bad state and accept the recovery. Please note that the sequence numbers will be set to zero during the recap.
ASK_RETRANS	See BID_RETRANS description above

Within each TABLE_INITPAINT message users will find one MD_TABLE_CMD_RT field/value pairing for the entire initial paint and then individual MD_TABLE_CMD_RT field/value pairings for each MBL_TABLE_ASK/MBO_TABLE_ASK/ MBL_TABLE_BID/MBO_TABLE_BID that may be present. Thereafter, users will see on MD_TABLE_CMD field/value pairing for each BID or ASK MKTDEPTH_EVENT_SUBTYPE tick update.

The possible string values, which indicate what action should be taken in response to the market depth event, are listed in the table below.

Name	Value	Description
UNASSIGNED	0	The default constant 'UNASSIGNED' is used to initialize all enumeration type fields
ADD	1	Add an entry to the order book. When this order is added in the market depth table, users should shift all orders at the market depth position in the event and market depth orders or levels inferior to event passed to one position inferior. For example, if a new order is added to position one of the market depth table, then the previous order at position one is shifted to position two. The order at position two is shifted to position three and so on until users get to the market depth window size. If the ADD results in Bid or ASK sides to have more levels than the value configured in MB[LO]_WINDOW_SIZE, the last level in the corresponding side should be dropped. It will be up to the user to cache MB[LO]_WINDOW_SIZE from the Initial paint event to handle this scenario.
DEL	2	Delete this event from the market depth cache. The delete should occur at the position passed in the market depth event. When cached market event at the position passed in the delete is removed, all position inferior should have their positions shifted by one. For example, if position one is deleted from a market by order or market by price event, the position two becomes one, position three becomes two, etc.
DELALL	3	Delete all events from the cache. This is a market depth flush usually passed at the start or end of trading or when a trading halt occurs.
DELBETTER	4	Delete this order and any superior orders. The order ID at the next inferior position is now the best order. This differs from the EXEC command in that it deletes the current order, where the EXEC command modifies the current order.
DELSIDE	5	Delete all events on the corresponding side (bid/ask) of the order book.
EXEC	7	Trade Execution. Find the corresponding order in the cache, replace event details with this event and then delete any prior superior orders.
MOD	8	Modify an existing event in the market depth cache. Find the cached market depth event by the position in the new market depth event and replace the cached event by the fields and data in the new event.
REPLACE	10	Replace previous price level or order at this position. Add price level or order if users do not have it currently in the cache. A zero (0) price and size will be sent when there is no active price or order at this level.
REPLACE_BY_BROKER	11	This table command is used for top of file feeds where the action is to replace by the broker mnemonic. The recipient needs to find the broker in their cache and replace the quote with the one in the market depth event. If that broker is not present, it should be added to the cache. If the price and size for a broker is set to 0, the broker should be deleted from the cache.

Name	Value	Description
CLEARALL	12	<p>Clears the entire orderbook for the specified side. This market depth table command is issued by Bloomberg when market depth recovery is occurring. This table command has the same effect on the cache as DELETEALL which means all order or levels should be cleared from the cache. During LVC recovery users will generally see 2 CLEARALLs - 1 for Bid side and 1 for Ask side. Should the client of market depth need to process a recovery of market depth differently, this table command allows the user to differentiate from the source/exchange produced DELETEALL.</p> <p>CLEARALL messages may occur without accompanying RETRANS labels in the event of data loss within Bloomberg network or upon the receipt of the first tick of a new trading day. Hence, upon the receipt of a CLEARALL, users should clear their book and prepare to receive the subsequent recover ADD messages.</p>
REPLACE_CLEAR	13	<p>The REPLACE_CLEAR table command is intended to remove an order or more often a level in the market depth cache. The REPLACE_CLEAR should be indexed by the MarketDepth.ByLevel/ByOrder.Bid/Ask.Position field. The cache should NOT be shifted up after the level is cleared. A clear means all orders at that position have been deleted from the order book. It is possible that an order or level at a superior or most superior position to be cleared prior to more inferior levels. After the level is cleared in this case, it is expected that subsequent market depth event(s) will be passed to clear the orders or levels at positions inferior to the one just cleared.</p>

The following code snippet demonstrates how to handle and print out a MarketDepth subscription to `std::cout`. This C++ snippet is based on the aforementioned "MarketDepthSubscriptionExample" C++ SDK example. For a more complete example that demonstrates how to handle and build an order/level book, please reference the aforementioned "MarketDepthSubscriptionSnapshotExample" example in either the Java, C++ or .NET SDK.

```
bool processEvent(const Event &event, Session *session)
{
    try {
        switch (event.eventType())
        {
            case Event::SUBSCRIPTION_DATA:
            {
                char timeBuffer[64];
                getTimeStamp(timeBuffer, sizeof(timeBuffer));
                std::cout << "Processing SUBSCRIPTION_DATA" << std::endl;
                MessageIterator msgIter(event);

                while (msgIter.next()) {
                    Message msg = msgIter.message();
                    std::string *topic = reinterpret_cast<std::string*>(
                        msg.correlationId().asPointer());

                    std::cout << timeBuffer << ": " << topic->c_str() << "
                        - " ; msg.print(std::cout);
                }
            }
        }
    }
}
```

```

        }
        break;
    }
    case Event::SUBSCRIPTION_STATUS:
        return
            processSubscriptionStatus(event);
        break;
    default:
        return
            processMiscEvents(event);
        break;
    }
} catch (Exception &e) {
    std::cout << "Library Exception !!! " << e.description().c_str() <<
        std::endl;
}return false;
}

```

Notice that the above code checks the EventType being returned and looks for SUBSCRIPTION_DATA. Please note that the processSubscriptionStatus() and processMiscEvents() functions were not shown for brevity. It is also noticed that the event handler for the tick updates is identical to that of a //blp/mktdata subscription, for instance.

6.1.5. Handling Multiple Messages (a.k.a. Fragments)

The summary (initial paint) messages can be split into one or more smaller messages in the case where the returned data is too large to fit into a single message. It will be up to the user to handle this in their application.

Users will achieve this by checking the Fragment type of any SUBSCRIPTION_DATA event message containing a MKTDEPTH_EVENT_SUBTYPE of value "TABLE_INITPAINT". The Fragment enum is used to indicate whether a message is a fragmented message or not and what position it occurs within the chain of split fragmented messages. If the TABLE_INITPAINT is split into two parts, then the first message will have a Fragment type value of FRAGMENT_START and a last message of FRAGMENT_END. If the TABLE_INITPAINT is split into more than 2 parts, all middle Fragments will be of type FRAGMENT_INTERMEDIATE.

This **enum** will exist in both MARKET_BY_ORDER and MARKET_BY_LEVEL messages.

Message::Fragment Type Enumerators	
FRAGMENT_NONE	Message is not fragmented
FRAGMENT_START	The first fragmented message
FRAGMENT_INTERMEDIATE	Intermediate fragmented messages
FRAGMENT_END	The last fragmented message

The following code snippet demonstrates how the C++ "MarketDepthSubscriptionSnapshotExample" example checks the fragment type. Please take a look at the full code example in the SDK for a working version of this code.

```

if (subType == TABLE_INITPAINT) {
    if (msg.fragmentType() ==
        BloombergLP::blpapi::Message::Fragment::FRAGMENT_START ||
        msg.fragmentType() ==
        BloombergLP::blpapi::Message::Fragment::FRAGMENT_NONE) {

        if (msg.hasElement(MBO_WINDOW_SIZE, true) ){
            d_orderBooks[Side::ASKSIDE].window_size = (unsigned
                int) msg.getElementAsInt64(MBO_WINDOW_SIZE);
            d_orderBooks[Side::BIDSIDE].window_size =
                d_orderBooks[Side::ASKSIDE].window_size;
        }
        d_orderBooks[Side::ASKSIDE].book_type =
            msg.getElementAsString(MD_BOOK_TYPE);
        d_orderBooks[Side::BIDSIDE].book_type =
            d_orderBooks[Side::ASKSIDE].book_type;

        // clear cache
        d_orderBooks[Side::ASKSIDE].doC
            learAll();
        d_orderBooks[Side::BIDSIDE].doC
            learAll();
    }
}

```

The above code checks the Market Depth Event Sub-Type being returned, and if it equals TABLE_INITPAINT, then it checks the Fragment Type. If a FRAGMENT_START or FRAGMENT_NONE type is returned by msg.fragmentType(), then the order book is cleared.

A. Data Response for ADD-MOD-DEL (AMD) Order Books

Every event in an Add-Mode-Delete (AMD) order book is critical in maintaining an accurate book. One missed message can result in a book that is wrong for the remainder of the trading day. Because of this, all AMD market depth messages have a MBO_SEQNUM_RT field with a non-zero value. This field is generated by the Bloomberg ticker plant when it creates its order book and increments monotonically for every update. The Sequence number is incremented per book. It is up to the user's application to clear the book as soon as they receive an initial paint message.

B. MBO-AMD sample subscription output

(for "//blp/mktdepthdata/bsym/CT/RIM?type=MBO").

```
Processing SUBSCRIPTION_DATA
MarketDepthUpdates = {
  MKTDEPTH_EVENT_TYPE = MARKET_BY_ORDER
  MKTDEPTH_EVENT_SUBTYPE = TABLE_INITPAINT
  ID_BB_SEC_NUM_SRC = 502511690826
  FEED_SOURCE = "CT"
  EID = 14184
  MD_TABLE_CMD_RT = ADD
  MD_BOOK_TYPE = MBO-AMD
  MBO_WINDOW_SIZE = 200
  MBL_TABLE_ASK[] = {
  }
  MBL_TABLE_BID[] = {
  }
  MBO_TABLE_ASK[] = {
    MBO_TABLE_ASK = {
      MBO_ASK_POSITION_RT = 1
      MBO_ASK_RT = 11.3199996948242
      MBO_ASK_BROKER_RT = "    1"
      MBO_ASK_COND_CODE_RT = ""
      MBO_ORDER_ID_RT =
      "3235323500004c1d0001" MBO_ASK_SIZE_RT
      = 200
      MBO_TIME_RT = 2012-05-25T19:53:06.000+00:00
      MD_TABLE_CMD_RT = ADD
    }
    MBO_TABLE_ASK = {
      MBO_ASK_POSITION_RT =
      2
      MBO_ASK_RT = 11.3199996948242
      MBO_ASK_BROKER_RT = "    1"
      MBO_ASK_COND_CODE_RT = ""
      MBO_ORDER_ID_RT =
      "3235323500004c1e0001" MBO_ASK_SIZE_RT
      = 100
      MBO_TIME_RT = 2012-05-25T19:53:06.000+00:00
      MD_TABLE_CMD_RT = ADD
    }
  }
```

```

    }
    ... (more)
    MBO_TABLE_BID[] = {
        MBO_TABLE_BID = {
            MBO_BID_POSITION_RT = 1
            MBO_BID_RT = 11.3100004196167
            MBO_BID_BROKER_RT = " 79"
            MBO_BID_COND_CODE_RT = ""
            MBO_ORDER_ID_RT =
                "32353235000075f8004f"
            MBO_BID_SIZE_RT = 1400
            MBO_TIME_RT = 2012-05-25T19:46:59.000+00:00
            MD_TABLE_CMD_RT = ADD
        }
        MBO_TABLE_BID = {
            MBO_BID_POSITION_RT =
                2
            MBO_BID_RT = 11.3100004196167
            MBO_BID_BROKER_RT = " 79"
            MBO_BID_COND_CODE_RT = ""
            MBO_ORDER_ID_RT =
                "323532350000761a004f"
            MBO_BID_SIZE_RT = 500
            MBO_TIME_RT = 2012-05-25T19:47:33.000+00:00
            MD_TABLE_CMD_RT = ADD
        }
        ... (more)
    }
    Processing SUBSCRIPTION_DATA
    MarketDepthUpdates = {
        MKTDEPTH_EVENT_TYPE = MARKET_BY_ORDER
        MKTDEPTH_EVENT_SUBTYPE = ASK
        EID = 14184
        MD_TABLE_CMD_RT = DEL
        MBO_SEQNUM_RT =
            199951
        MBO_ASK_POSITION_RT = 7
        MBO_ASK_RT = 11.3199996948242
        MBO_ASK_BROKER_RT = " 79"
        MBO_ASK_COND_CODE_RT = ""
        MBO_ORDER_ID_RT =
            "323532350000774e004f"
        MBO_ASK_SIZE_RT = 500
        MBO_TIME_RT = 2012-05-25T19:53:55.000+00:00
        MBL_TABLE_ASK[] = {
        }
        MBL_TABLE_BID[] = {
        }
        MBO_TABLE_ASK[] = {
        }
        MBO_TABLE_BID[] = {
        }
    }
}

```

```

Processing SUBSCRIPTION_DATA
/bsym/CT/RIM - MarketDepthUpdates = {
  MKTDEPTH_EVENT_TYPE = MARKET_BY_ORDER
  MKTDEPTH_EVENT_SUBTYPE = TABLE_INITPAINT
  ID_BB_SEC_NUM_SRC = 502511690826
  FEED_SOURCE = "CT"
  EID = 14184
  MD_TABLE_CMD_RT = ADD
  MD_BOOK_TYPE = MBO-AMD
  MBO_WINDOW_SIZE = 200
  MBL_TABLE_ASK[] = {
  }
  MBL_TABLE_BID[] = {
  }
  MBO_TABLE_ASK[] = {
    MBO_TABLE_ASK = {
      MBO_ASK_POSITION_RT = 200
      MBO_ASK_RT = 12
      MBO_ASK_BROKER_RT = "
                                8
      0" MBO_ASK_COND_CODE_RT
      = ""
      MBO_ORDER_ID_RT = "3235313500000c390050"
      MBO_ASK_SIZE_RT = 100
      MBO_TIME_RT = 2012-05-25T15:20:49.000+00:00
      MD_TABLE_CMD_RT = ADD
    }
  }
  MBO_TABLE_BID[] = {
  }
}

```

Notes:

The first message above is the initial paint (as indicated by the TABLE_INITPAINT event sub-type (i.e., MKTDEPTH_EVENT_SUBTYPE)) and indicates that it is a Market-By-Order message, as indicated by the MARKET_BY_ORDER event type (i.e., MKTDEPTH_EVENT_TYPE). Within the initial paint message, users will find a table of asks and bids. In this case, it is an MBO request, so the table will be of MBO bids and asks (indicated by MBO_TABLE_BID[] and MBO_TABLE_ASK[] array items). When users receive an initial paint message, they should clear their book prior to populating with the table of Asks and Bids.

Because this is an AMD (Add-Mod-Del) MBO Book Type, the MD_TABLE_CMD_RT field in the initial paint is ADD. The valid table commands for subsequent AMD type message updates are ADD, MOD, DELETE and CLEARALL.

C. Data Response for Request-By-Broker (RBB) Order Books

Because the Replace-By-Broker (RBB) method addresses individual broker orders, it applies only to MBO order books. Unlike AMD and RBP, there is no concept of row numbers in an RBB order book. Instead each broker ID represents a row. This leaves it up to the feed handler to decide how to order the book. Typically they are ordered by best (highest) bid and best (lowest) ask to worst (lowest) bid and worst (highest) ask. If multiple orders exist at the same price on the same side then they can be sorted by size or by broker code. It is up to the user's application to clear the book as soon as they receive an initial paint message.

D. MBO-RBB Subscription Output

(for `"//blp/mktdepthdata/bsym/US/AAPL?type=MBO"`)

```
Processing SUBSCRIPTION_DATA
MarketDepthUpdates = {
  MKTDEPTH_EVENT_TYPE = MARKET_BY_ORDER
  MKTDEPTH_EVENT_SUBTYPE =
  TABLE_INITPAINT ID_BB_SEC_NUM_SRC =
  399432471918 FEED_SOURCE = "US"
  EID = 14023
  MD_TABLE_CMD_RT =
  REPLACE_BY_BROKER MD_BOOK_TYPE =
  MBO-RBB MBL_TABLE_ASK[] = {
  }
  MBL_TABLE_BID[] = {
  }
  MBO_TABLE_ASK[] = {
    MBO_TABLE_ASK = {
      MBO_ASK_RT = 604.630126953125
      MBO_ASK_BROKER_RT = "ADAM"
      MBO_ASK_BROKER_MODE_RT = OPEN
      MBO_ASK_COND_CODE_RT = ""
      MBO_ASK_COND_CODE_SRC_RT = ""
      MBO_ASK_LSRC_RT = "UQ"
      MBO_ASK_SIZE_RT = 100
      MBO_TIME_RT = 2012-05-25T13:44:01.000+00:00
      MD_TABLE_CMD_RT = REPLACE_BY_BROKER
    }
    MBO_TABLE_ASK = {
      MBO_ASK_RT =
      560.75
      MBO_ASK_BROKER_RT = "ARCX"
      MBO_ASK_BROKER_MODE_RT = OPEN
      MBO_ASK_COND_CODE_RT = ""
      MBO_ASK_COND_CODE_SRC_RT = ""
      MBO_ASK_LSRC_RT = "UP"
      MBO_ASK_SIZE_RT = 200
      MBO_TIME_RT = 2012-05-25T19:24:12.000+00:00
      MD_TABLE_CMD_RT = REPLACE_BY_BROKER
    }
    ... (more)
  }
  MBO_TABLE_BID[] = {
    MBO_TABLE_BID = {
      MBO_BID_RT = 514.900146484375
      MBO_BID_BROKER_RT = "ADAM"
      MBO_BID_BROKER_MODE_RT = OPEN
      MBO_BID_COND_CODE_RT = ""
      MBO_BID_COND_CODE_SRC_RT = ""
      MBO_BID_LSRC_RT = "UQ"
      MBO_BID_SIZE_RT = 100
      MBO_TIME_RT = 2012-05-25T13:44:01.000+00:00
      MD_TABLE_CMD_RT = REPLACE_BY_BROKER
    }
  }
```



```
        MBO_TABLE_BID = {
            MBO_BID_RT = 560.60009765625
            MBO_BID_BROKER_RT = "ARCX"
            MBO_BID_BROKER_MODE_RT = OPEN
            MBO_BID_COND_CODE_RT = ""
            MBO_BID_COND_CODE_SRC_RT = ""
            MBO_BID_LSRC_RT = "UP"
            MBO_BID_SIZE_RT = 200
            MBO_TIME_RT = 2012-05-25T19:24:13.000+00:00
            MD_TABLE_CMD_RT = REPLACE_BY_BROKER
        }
        ... (more)
    }
}
Processing SUBSCRIPTION_DATA
MarketDepthUpdates = {
    MKTDEPTH_EVENT_TYPE = MARKET_BY_ORDER
    MKTDEPTH_EVENT_SUBTYPE = BID
    EID = 14023
    MD_TABLE_CMD_RT = REPLACE_BY_BROKER
    MBO_TIME_RT = 2012-05-
    25T19:24:14.000+00:00 MBO_BID_RT =
    560.56005859375 MBO_BID_BROKER_RT =
    "NQBX" MBO_BID_BROKER_MODE_RT = OPEN
    MBO_BID_COND_CODE_RT = ""
    MBO_BID_COND_CODE_SRC_RT = ""
    MBO_BID_LSRC_RT = "UB"
    MBO_BID_SIZE_RT = 100
    MBL_TABLE_ASK[] = {
    }
    MBL_TABLE_BID[] = {
    }
    MBO_TABLE_ASK[] = {
    }
    MBO_TABLE_BID[] = {
    }
}
```

```

Processing SUBSCRIPTION_DATA
MarketDepthUpdates = {
    MKTDEPTH_EVENT_TYPE = MARKET_BY_ORDER
    MKTDEPTH_EVENT_SUBTYPE = BID
    EID = 14023
    MD_TABLE_CMD_RT = REPLACE_BY_BROKER
    MBO_TIME_RT = 2012-05-
    25T19:24:14.000+00:00 MBO_BID_RT =
    560.60009765625 MBO_BID_BROKER_RT = "ARCX"
    MBO_BID_BROKER_MODE_RT = OPEN
    MBO_BID_COND_CODE_RT = ""
    MBO_BID_COND_CODE_SRC_RT = ""
    MBO_BID_LSRC_RT = "UP"
    MBO_BID_SIZE_RT = 100
    MBL_TABLE_ASK[] = {
    }
    MBL_TABLE_BID[] = {
    }
    MBO_TABLE_ASK[] = {
    }
    MBO_TABLE_BID[] = {
    }
}

```

Notes:

The first message above is the initial paint (as indicated by the TABLE_INITPAINT event sub-type (i.e., MKTDEPTH_EVENT_SUBTYPE)) and indicates that it is a Market-By-Order message, as indicated by the MARKET_BY_ORDER event type (i.e., MKTDEPTH_EVENT_TYPE). Within the initial paint message, users will find a table of asks and bids. In this case, it is an MBO request, so the table will consist of MBO bids and asks (indicated by MBO_TABLE_BID[] and MBO_TABLE_ASK[] array items). When users receive an initial paint message, they should clear their book prior to populating with the array of Asks and Bids.

Because this is a Request-By-Broker (RBB) MBO Book Type, the MD_TABLE_CMD_RT field in the initial paint and subsequent update is REPLACE_BY_BROKER. The other valid table commands for an RBB type are REPLACE_CLEAR and CLEARALL, which are sent by the exchange.

E. Data Response for Request-By-Position (RBP) Order Books

With the Replace-By-Position (RBP) method each level is explicitly sent so that to maintain the order book the feed handler simply has to apply the data for each level directly. There is no shifting of rows in the order book. Because each level is maintained individually (unlike the AMD method) missed messages, while never a good thing, have no impact other than that they were missed. All other levels retain their correct values.

The RBP method is generally easier to implement than AMD, but this comes with a cost. Because each level is maintained individually a new value at level one requires that the entire order book be resent. The bandwidth impact for small order books is minimal but can be extreme for large order books. For this reason AMD is often used for large order books.

F. MBL-RBP Subscription Output

(for “//blp/mktdepthdata/ticker/ESM2 Index?type=MBL”).

```
Processing SUBSCRIPTION_DATA
MarketDepthUpdates = {
  MKTDEPTH_EVENT_TYPE = MARKET_BY_LEVEL
  MKTDEPTH_EVENT_SUBTYPE = TABLE_INITPAINT
  ID_BB_SEC_NUM_SRC = 2078784978839
  FEED_SOURCE = "eCME"
  EID = 14002
  MD_TABLE_CMD_RT = REPLACE
  MD_BOOK_TYPE = MBL-RBP
  MBL_WINDOW_SIZE = 10
  MBL_TABLE_ASK[] = {
    MBL_TABLE_ASK = {
      MBL_ASK_POSITION_RT = 1
      MBL_ASK_RT = 1314.75
      MBL_ASK_COND_CODE_RT = ""
      MBL_ASK_NUM_ORDERS_RT = 35
      MBL_ASK_SIZE_RT = 384
      MBL_TIME_RT = 2012-05-25T20:05:13.302+00:00
      MD_TABLE_CMD_RT = REPLACE
    }
    MBL_TABLE_ASK = {
      MBL_ASK_POSITION_RT = 2
      MBL_ASK_RT = 1315
      MBL_ASK_COND_CODE_RT = ""
      MBL_ASK_NUM_ORDERS_RT = 65
      MBL_ASK_SIZE_RT = 397
      MBL_TIME_RT = 2012-05-25T20:05:13.648+00:00
      MD_TABLE_CMD_RT = REPLACE
    }
    ... (more)
  }
  MBL_TABLE_BID[] = {
    MBL_TABLE_BID = {
      MBL_BID_POSITION_RT = 1
      MBL_BID_RT = 1314.5
      MBL_BID_COND_CODE_RT = ""
      MBL_BID_NUM_ORDERS_RT = 65
      MBL_TIME_RT = 2012-05-25T20:05:13.043+00:00
      MBL_BID_SIZE_RT = 427
      MD_TABLE_CMD_RT = REPLACE
    }
    MBL_TABLE_BID = {
      MBL_BID_POSITION_RT = 2
      MBL_BID_RT = 1314.25
      MBL_BID_COND_CODE_RT = ""
      MBL_BID_NUM_ORDERS_RT = 69
      MBL_TIME_RT = 2012-05-25T20:05:11.351+00:00
      MBL_BID_SIZE_RT = 631
      MD_TABLE_CMD_RT = REPLACE
    }
  }
}
```

```

    }
    ... (more)
  }
}
Processing SUBSCRIPTION_DATA
MarketDepthUpdates = {
  MKTDEPTH_EVENT_TYPE = MARKET_BY_LEVEL
  MKTDEPTH_EVENT_SUBTYPE = ASK
  EID = 14002
  MD_TABLE_CMD_RT = REPLACE
  MD_MULTI_TICK_UPD_RT = 0
  MBL_ASK_POSITION_RT = 2
  MBL_ASK_RT = 1315
  MBLASK_COND_CODE_RT = ""
  MBL_ASK_NUM_ORDERS_RT = 66
  MBL_ASK_SIZE_RT = 398
  MBL_TIME_RT = 2012-05-25T20:05:14.085+00:00
  MBL_TABLE_ASK[] = {
  }
  MBL_TABLE_BID[] = {
  }
  MBO_TABLE_ASK[] = {
  }
  MBO_TABLE_BID[] = {
  }
}
Processing SUBSCRIPTION_DATA
MarketDepthUpdates = {
  MKTDEPTH_EVENT_TYPE = MARKET_BY_LEVEL
  MKTDEPTH_EVENT_SUBTYPE = ASK
  EID = 14002
  MD_TABLE_CMD_RT = REPLACE
  MD_MULTI_TICK_UPD_RT = 0
  MBL_ASK_POSITION_RT = 2
  MBL_ASK_RT = 1315
  MBL_ASK_COND_CODE_RT = ""
  MBL_ASK_NUM_ORDERS_RT = 65
  MBL_ASK_SIZE_RT = 397
  MBL_TIME_RT = 2012-05-25T20:05:14.148+00:00
  MBL_TABLE_ASK[] = {
  }
  MBL_TABLE_BID[] = {
  }
  MBO_TABLE_ASK[] = {
  }
  MBO_TABLE_BID[] = {
  }
}

```

Notes:

The first message above is the initial paint (as indicated by the TABLE_INITPAINT event sub-type (i.e. MKTDEPTH_EVENT_SUBTYPE)) and indicates that it is a Market-By-Level (MBL) message, as indicated by the MARKET_BY_LEVEL event type (i.e.

MKTDEPTH_EVENT_TYPE).

Within the initial paint message, users will find the MBL_WINDOW_SIZE. This indicates the number of levels in the book, along with the table command (i.e. MD_TABLE_CMD_RT) with a value of "REPLACE" and book type (i.e.

MD_BOOK_TYPE) with a value of "MBL-RBP".

Because this is a Request-By-Position (RBP) MBL Book Type, the MD_TABLE_CMD_RT field in the initial paint is REPLACE and all subsequent updates will possess a table command of either REPLACE_CLEAR, REPLACE or CLEARALL. This is true for both MBO and MBL event types. The output above includes a sample BID/REPLACE and ASK/ REPLACE_CLEAR message.

G. Order Book Recaps

Order book recaps provide all the information required to completely rebuild an order book. They can be initiated by the exchange, B-PIPE or the client application.

Recaps apply to every style of order book: Add-Mod-Delete (AMD), Replace-by-Position (RBP) and Replace-by-Broker (RBB), but they play a special role for AMD order books. It is critical that AMD order books receive every message. A single missed message (a data gap) can result in the AMD book being wrong for the remainder of the market day. RBP and RBB books tend to be self-correcting in the event of a data gap making gap detection less critical.

The MBL_SEQNUM_RT and MBL_SEQNUM_RT fields are sequentially increasing numbers included only in AMD order book market depth messages. They allow the client application to detect gaps in the AMD market depth messages. A sequence number 5 followed by 7 indicates that a gap of one message occurred.

H. Gap Detection

Data gaps occur as a result of missed network messages. While rare, as in every complex networked system, missed messages can occur at any level and for many reasons. If a data gap occurs between the B-PIPE order book systems and the application, it is the client application's responsibility to take action to restore the order book to an accurate state. If the gap is detected by the Bloomberg upstream order book systems, B-PIPE will automatically initiate the recap without any action by the client application.

When B-PIPE detects a gap in the MBL or MBO "AMD" order book, the MD_GAP_DETECTED field is present and set to "true" in every market depth update message for each effected order book. This informs the client application that B-PIPE has detected the gap and to expect an automatic recap.

MD_GAP_DETECTED will not be present once the recap is sent. Therefore, even though a client application detects a gap, if this field is present in market depth update messages, no further action is required by the client application except to begin reading the recap messages, which will follow immediately and be indicated with a MKTDEPTH_EVENT_SUBTYPE of BID_RETRANS and ASK_RETRANS in each message update. In cases where a sequence number gap is detected but the MD_GAP_DETECTED field is not present in the message, it is the responsibility of the client application to request a recap (i.e., resubscribe) to the order book.

Fields Affected by Recaps

Fields	Descriptions
MKTDEPTH_EVENT_SUBTYPE	Present in every market depth message for all styles of orderbook. When an unsolicited recap is in progress, this field will have a value of "BID_RETRANS" or "ASK_RETRANS".
MBL_SEQNUM_RT and MBO_SEQNUM_RT	Present in every market depth message for AMD, and only AMD, order books. They will have a value of 0 if the message is part of an order book recap, regardless of how initiated. Gap detection does not apply to recaps. The value of these fields in the first non-recap market depth update message following the recap will have a non-zero value which should be used to detect any gaps following the recap.
MD_TABLE_CMD_RT	Present in every market depth message, it indicates the action to take for this market depth message. The behavior of this field is unchanged. A value of "DELSIDE" indicates that the appropriate side of the order book (bid or ask) should be cleared of all values. All recaps start with a DELSIDE. All other values should be applied as already documented.

Fields Affected by Recaps

Fields	Descriptions
MD_MULTI_TICK_UPD_RT	When present, indicates that a market depth message is one of multiple messages that make up a single update to an order book. A value of 1 indicates that additional market depth messages that are part of the same order book update will follow this message. A value of 0 indicates that this is the last message in the update and that the update is complete. All recaps for every style of order book are sent as multi-tick updates. Multi-tick updates may also be used to send non-recap RBP style order book updates.

6.2. Market List Service (//blp/mktlist)

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 snapshot 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 Non- BPS 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 snapshot 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", "GDCO List" or "Yield Curve". See table below for additional information and examples of each.
	/secids	Snapshot 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
	/gdco	For Requests by GDCO syntax
	/bpkbl	Requests by Bloomberg parsekeyable Identifier
	/esym	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
		/bbgid
	The following topic types do not require a source and consist of value alone <Identity>	/bsid
		/eid
		/ticker
	The following topic type consists of only a <source>	/source
	The following topic type consists of Broker ID and Mon ID separated by the forward slash. <broker_id>/<mon_id>	/gdco

 For additional information on refer to the "Reference Services and Schemas Guide".

6.2.1. Code Examples

Users will find two separate examples 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" snapshot request and displays the message to the console window.

Now that users have a better understanding as to how a `//blp/mktlist` subscription or snapshot string is formed, it is now time to use it in their application. The following sections provide further details as to how to subscribe to a chain of instruments and request a Snapshot of a list of members.

6.2.2. Subscribing To Instrument Chains

Overview

B-PIPE supports the ability to subscribe to lists of instruments known as chains. When a subscription is made for a chain, the request must first resolve to a single B-PIPE instrument. This instrument is called the "underlying instrument".

The instruments returned in the list are referred to as "list members". The characteristics of list members depend upon the security class of the underlying instrument or parameters included in the initial chain request. Examples are list members that are options or members that are futures.

In most cases, the list members will all be the same security class. When the underlying security class is an Index or Curve, the security class of the each member may or may not be same.

In most cases, underlying instruments are regular B-PIPE instruments, such as an equity or futures contract. Other times, the underlying instrument will be a pseudo instrument whose sole purpose is to serve as the underlying instrument for the chain. Like all other instruments on B-PIPE, the underlying pseudo-instrument has its own, unique `ID_BB_SEC_NUM_SRC`. It can be subscribed to as a regular instrument but since it has no price data of its own the subscription will only return reference data.

For most chains, the relationship between the underlying instrument and the list members is established by the B-PIPE service when the subscription is made using the BSID of the underlying instrument. Every member of the list has a `LIST_UNDERLYING_ID_BSID` field, which contains the BSID value of the underlying instrument, and all matching instruments of the appropriate security class are returned in the list of members.

Index and Curve lists are handled differently. The list's members are maintained by the Bloomberg Data Center. Once it is determined that this list subscription is for index or curve members, the Bloomberg Data Center is queried for the list of members. This list contains the terminal ticker (ParseKeyable symbol) for each member, which is resolved to an instrument on B-PIPE. It is possible that an index or curve list member is not available on B-PIPE. In this case, the list member will be included in the list, but return only the ParseKeyable symbol.

This allows the requestor to contact Bloomberg about getting the missing instrument added to B-PIPE.

The default security class of the list members depends on the security class of the underlying instrument specified in the request. The default can be overridden using the optional parameter "secclass". The table below defines the default security class of the list members for each underlying instrument security class.

Underlying Security Class	Default Chain Member Security Class
Currency	Option
Equity	Option
Fixed Income	N/A
Fund	Option
Future Root	Future
Future Contract	Option
Index	Members
Option	N/A
Warrant	N/A
Curve	Members

An alternate security class for the returned members is available and can be specified in the subscription string using a parameter. For example, the following chain requests are equivalent because the default member security class is Option:

```
//blp/mktlist/chain/bsym/US/IBM
```

```
//blp/mktlist/chain/bsym/US/IBM;secclass=Option
```

However, by using a parameter, a list of Futures with IBM can be obtained as the underlying instrument:

```
//blp/mktlist/chain/bsym/US/IBM;secclass=Future
```

In order to further qualify the subscription string, a parameter "source" can be applied. The value of this parameter is assigned by the user or application to limit the amount of returned members to those belonging to the specified source(s) only. More than one value is allowed for this parameter.

The "source" can be substituted by a "~". This value can be used when the client assumes that there is only one source for the security and there is no actual need to specify it. If this is the case, the subscription request will be processed successfully, but if the security has more than one source and the request is ambiguous, then the client will receive a SubscriptionFailure response with a NOTUNIQUE description. An example of such a subscription string would be "`//blp/mktlist/chain/cusip/~/459200101`".

6.2.3. Chain Subservice Examples

Type of Chain List	Example Subscription String	Topic Type	Topic Keya	Refreshesb
Option Chains	<code>//blp/mktlist/chain/bsym/LN/VOD</code>	<code>/bsym</code>	<code>/<DX282>/<DY003></code>	No
	<code>//blp/mktlist/chain/bsid/678605358297</code>	<code>/bsid</code>	<code>/<ID122></code>	No
	<code>//blp/mktlist/chain/buid/LN/EQ0010160500001000</code>	<code>/buid</code>	<code>/<DX282>/<ID059></code>	No
	<code>//blp/mktlist/chain/bbid/LN/EQ0010160500001000</code>	<code>/bbid</code>	<code>/<DX282>/<ID059></code>	No
	<code>//blp/mktlist/chain/bpkbl/VOD LN Equity</code>	<code>/bpkbl</code>	<code>/<DX194></code>	No
	<code>//blp/mktlist/chain/esym/LN/VOD</code>	<code>/esym</code>	<code>/<DX282>/<EX005></code>	No
	<code>/blp/mktlist/chain/cusip/UN/459200101</code>	<code>/cusip</code>	<code>/<DX282>/<ID032></code>	No
	<code>//blp/mktlist/chain/isin/LN/GB00BH4HKS39</code>	<code>/isin</code>	<code>/<DX282>/<ID005></code>	No
	<code>//blp/mktlist/chain/sedol/LN/BH4HKS3</code>	<code>/sedol</code>	<code>/<DX282>/<ID002></code>	No

Type of Chain List	Example Subscription String	Topic Type	Topic Keya	Refreshesb
	//blp/mktlist/chain/bbgid/LN/BBG000C6K6G9	/bbgid	/<DX282>/<ID135>	No
	//blp/mktlist/chain/ticker/VOD LN Equity	/ticker	/<DX194>	No
Index List	//blp/mktlist/chain/bsym/FTUK/UKX	/bsym	/<DX282>/<DY003>	Daily
Yield Curve	//blp/mktlist/chain/bpkbl/YCMM0010 Index	/bpkbl	/<identifier>	Daily
GDCO	//blp/mktlist/chain/gdco/broker/id	/gdco	/<broker_id>/<mon_id>	N/A
EID List	//blp/mktlist/chain/eid/14014	/eid	/<source>	No
Source List	//blp/mktlist/chain/source/UN;secclass=Equity	/source	/<source>	No

- The **FLDS** <GO> identifier associated with the expected key values for that particular topic is listed, where applicable, which can be found on **FLDS** <GO> on the Bloomberg Professional service
- 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.

Here is a quick reference for the above **FLDS** <GO> identifiers:

FLDS <GO> Identifier	Mnemonic	FLDS <GO> Identifier	Mnemonic
DX194	PARSEKYABLE_DES_SOURCE	ID005	ID_ISIN
DX282	FEED_SOURCE	ID032	ID_CUSIP
DY003	ID_BB_SEC_NUM_DES	ID059	ID_BB_UNIQUE
EX005	ID_EXCH_SYMBOL	ID122	ID_BB_SEC_NUM_SRC
ID002	ID_SEDOL1	ID035	ID_BB_GLOBAL

Additional "chain" Subscription Examples

Subscription String	Returns
//blp/mktlist/chain/bsym/FTUK/UKX Index;secclass=Option	Returns options on the UKX Index
//blp/mktlist/chain/bsym/FTUK/UKX	Returns options on the UKX Index traded on source
//blp/mktlist/chain/cusip/~/.459200101	SubscriptionFailure: ErrorCode=2; Description=NOTUNIQUE; Category=BAD_SEC Note: NOTUNIQUE is returned because the security has more than one source and the request is ambiguous.
//blp/mktlist/chain/bsid/1086627109973	Options for IBM Equity
//blp/mktlist/chain/bsym/US/IBM;secclass=Future	Returns futures for Equity
//blp/mktlist/chain/bpkbl/YCMM0010 Index	GBP LIBOR Curve members (Yield Curve)
//blp/mktlist/chain/eid/38736	List of all currencies available on EID 38736
//blp/mktlist/chain/bsym/US/HP	Returns a chain of options for the composite equity
//blp/mktlist/chain/bsym/DJI/INDU Index	Returns a chain of the members of the index.
//blp/mktlist/chain/bsid/1086627109973	This resolves to currency (/IT/UBY) so will return an option chain.
//blp/mktlist/chain/isin/LN/GB00B16GWD56;secclass=Warrant	Returns a chain of warrants for the underlying instrument.
//blp/mktlist/chain/bsym/FTUK/UKX Index;secclass=Index	Returns a chain of members for the specified index identifier (equivalent to //blp/mktlist/chain/bsym/FTUK/UKX Index)

Subscription String	Returns
//blp/mktlist/chain/source/UN;secclass=Equity	Returns a list of Equities under source UN
//blp/mktlist/chain/bsym/BGN/YCCF0009 Index	Returns the list of members for the curve "YCCF0009 Index"
//blp/mktlist/chain/bsid/1086627109973	This resolves to currency (/IT/UBY) so will return an option chain.
//blp/mktlist/chain/bpkbl/IBM US Equity	Returns a chain of options (equivalent to //blp/mktlist/chain/bsid/399432473346; secclass=Option).
//blp/mktlist/chain/isin/LN/GB00B16GWD56;secclass=Warrant	Returns a chain of warrants for the underlying instrument.
//blp/mktlist/chain/bsym/eNYL/XG1;secclass=Future	Returns a chain of futures for the underlying

The following code snippet demonstrates how to subscribe for streaming market list chain data and assumes that a session already exists and that the "//blp/mktlist" service has been successfully opened.

```
const char *security = "//blp/mktlist/chain/esym/LN/BP";
SubscriptionList subscriptions;
subscriptions.add(security, CorrelationId((char
*)security)); session.subscribe (subscriptions);
```

Response Overview

The Market List response will be a series of SUBSCRIPTION_DATA events, which users will be familiar with if they have developed Bloomberg API applications using any of the other streaming services, such as //blp/mktdata, //blp/mktwap or //blp/mktdepthdata.

A SUBSCRIPTION_DATA event message will either be of type ListRecap or ListData. The initial such event message(s) will be of type ListRecap. These represent the initial point of the chain of instruments. Within a single ListRecap message, users will find a LIST_LISTTYPE, comprising zero, or more, LIST_INSERT_ENTRIES.

If a subscription is made for a chain that does not contain any members, an empty list will be returned. An example of this is requesting the options for an equity that does not have any options. Although, there are no options for the equity, the subscription succeeds and a single ListRecap message will be received with LIST_INSERT_ENTRIES[] showing no elements. If the LIST_MUTABLE field value, from the ListRecap message is equal to 'MUTABLE', then that means there could be ListData items received later on, so users may wish to keep the subscription alive. The newly created members are then added to the previously empty list. However, if the LIST_MUTABLE field is 'IMMUTABLE', then that means it will not return any further updates and users may wish to terminate the subscription by unsubscribing. This is explained further in the following paragraph.

Various types of lists are available for a subscription. Though the subscription formats are the same, the lists could be:

ORDERED	When a list is subscribed and the LIST_ORDERED field within the ListRecap message equals 'ORDERED', the items on the list are returned in ordered format.
UNORDERED	When a list is subscribed and the LIST_ORDERED field within the ListRecap message equals 'NOTORDERED', the returned list of instruments could be in any order.

Similarly, a list subscription can be:

MUTABLE	If the LIST_MUTABLE field within the ListRecap message equals 'MUTABLE', the constituent instruments of a list can change. All subsequent updates will be received as ListData messages.
IMMUTABLE	If the LIST_MUTABLE field within the ListRecap message equals 'IMMUTABLE', the list of instruments will never change.

6.2.4. List Actions

ListAction	Description
CLEAR	Delete all existing list members. This implies there is more data to come
ADD	Add all of the list members in this set
CLEAR_AND_ADD	Delete all of the existing list members and then Add all of the list members in this sequence
DELETE	Delete all of the list members in this set. Member Identifiers much match the current Member Identifiers exactly
END	This is the last set in the sequence.
CLEAR_AND_END	Delete all of the existing list members, as there are no more entries to follow (i.e. the list is empty)
ADD_AND_END	Add all of the list members in this set and end. There are no more entries in this sequence
CLEAR_AND_ADD_AND_END	Delete all of the existing list members, add this entry and end. There are no more entries in this sequence.
DELETE_AND_END	Delete all of the list members in this set. Identifiers much match the current Member Identifiers exactly. Then end, as there are no more entries in this sequence.

6.2.5. Data Response For a "chain" Subscription

Here is sample Market List Chain output (A few entries from the beginning and end of a ListRecap message, along with one ListData message) for a Market List subscription to "// blp/mktlist/chain/source/TQ":

```
ListRecap = {
  LIST_ID =
  //blp/mktlist/chain/source/TQ EID
  = 35009

  LIST_LISTTYPE = Source List
  LIST_INSERT_ENTRIES[] =
    LIST_INSERT_ENTRIES = {
      ID_BB_SEC_NUM_SRC =
      7992941317759 FEED_SOURCE
      = TQ ID_BB_SEC_NUM_DES =
      RHI ID_BB_UNIQUE =
      EQ0000000006685436
      SECURITY_TYP2 = Equity
    }

    LIST_INSERT_ENTRIES = {
      ID_BB_SEC_NUM_SRC =
      7992941317760 FEED_SOURCE
      = TQ ID_BB_SEC_NUM_DES =
```

```

        GIL ID_BB_UNIQUE =
        EQ0000000006687052
        SECURITY_TYP2 = Equity
    }

    LIST_INSERT_ENTRIES = {
        ID_BB_SEC_NUM_SRC =
        7992961685384 FEED_SOURCE =
        TQ ID_BB_SEC_NUM_DES = ECONB
        ID_BB_UNIQUE =
        EQ0000000023559102
        SECURITY_TYP2 = Equity
    }

    LIST_INSERT_ENTRIES = {
        ID_BB_SEC_NUM_SRC =
        7992961685385 FEED_SOURCE =
        TQ ID_BB_SEC_NUM_DES = FIS1V
        ID_BB_UNIQUE =
        EQ0000000023561882
        SECURITY_TYP2 = Equity
    }

    LIST_INSERT_ENTRIES = {
        ID_BB_SEC_NUM_SRC =
        7992961842174 FEED_SOURCE =
        TQ ID_BB_SEC_NUM_DES = ENQ1
        ID_BB_UNIQUE =
        EQ0000000023716301
        SECURITY_TYP2 = Equity
    }

    LIST_ORDERED = NOTORDERED
    LIST_MUTABLE = MUTABLE
}

ListData = {
    LIST_ID =
    //blp/mktlist/chain/source/TQ EID =
    35009

    LIST_ACTION =
    ADD_AND_END FEED_SOURCE
    = TQ ID_BB_SEC_NUM_DES
    = SNOP
}

```

In the above sample output, a ListRecap message was returned first with a large number of list entries (only the partial recap is shown, however) and a single ListData message, which is an actual update to the subscription. Although, the ListRecap does not possess a LIST_ACTION value, users are to treat such a message as a CLEAR_AND_ADD action. In other words, the user will clear their cache and add the entries included in the message.

In the ListRecap message, users will notice a few other pieces of information in addition to the entries, such as the LIST_LISTTYPE field (in this case, its value is "Source List", which they will find included in the "TABLE OF SUBSERVICE NAME EXAMPLES" shown earlier in this section), the EID and the LIST_MUTABLE value, which is MUTABLE in this case. This indicates that the constituent instruments of a list can change.

After the ListRecap message, users will see one such change to the list, which is returned in the form of a ListData message. This message includes the LIST_ACTION, among other fields. In this case, it is indicating that they will ADD this message to their list at the END (as indicated by ADD_AND_END).

6.2.6. Handling Multiple Messages (a.k.a. Fragments)

The summary (initial paint) messages can be split into one or more smaller messages in the case where the returned data is too large to fit into a single message. It will be up to the user to handle this in their application.

Users will achieve this by checking the Fragment type of any SUBSCRIPTION_DATA event ListRecap message. The Fragment enum is used to indicate whether a message is a fragmented message or not and what position it occurs within the chain of split fragmented messages. If the ListRecap is split into two parts, then the first message will have a Fragment type value of FRAGMENT_START and a last message of FRAGMENT_END. If the ListRecap is split into more than 2 parts, all middle Fragments will be of type FRAGMENT_INTERMEDIATE. Message::Fragment Type Enumerators

Enumerator	Description
FRAGMENT_NONE	Message is not fragmented
FRAGMENT_START	The first fragmented message
FRAGMENT_INTERMEDIATE	Intermediate fragmented messages
FRAGMENT_END	The last fragmented message

To check for the Fragment Type, users will call the fragmentType property of the Message object (e.g. msg.fragmentType()). Within their application, they will check to see if the fragment type of the ListRecap message is FRAGMENT_NONE or FRAGMENT_START. If one of these is determined, then users will want to clear their list and begin adding the entries included in that part of the ListRecap message. In the case where FRAGMENT_START is determined, then they will know to continue reading the ListRecap messages and adding the entries to their list from those messages until they receive a ListRecap with a fragment type for FRAGMENT_END. At this point, users are indicated that they have finished building their list and it is now time to wait for any subsequent ListData updates.

6.2.7. Snapshot Request for List of Security Identifiers

If users would like to retrieve a list of all available sources that are pricing a given instrument, then they will use the 'secids' subservice. This request is particularly useful when the original subscription string provided by the client triggers a 'NOTUNIQUE' response from the service. Using this subservice, users also have the ability to filter their results to only a particular source.

The following table lists all of the supported Topic Types, their applicable topic key formats and associated B-PIPE mnemonic and **FLDS** <GO> field identifiers.

Topic Type	Topic Key	B-PIPE Field	FLDS<GO> Field
/bpkbl	/<identifier>	PARSEKYABLE_DES_SOURCE	DX194 and DS587
/bsid	/<identifier>	ID_BB_SEC_NUM_SRC	ID122
/bsym	/<identifier>	ID_BB_SEC_NUM_DES	DY003
/buid	/<identifier>	ID_BB_UNIQUE	ID059
/cusip	/<identifier>	ID_CUSIP	ID032
/esym	/<identifier>	ID_EXCH_SYMBOL	EX005->EX011

Topic Type	Topic Key	B-PIPE Field	FLDS<GO> Field
/isin	/<identifier>	ID_ISIN	ID005
/sedol	/<identifier>	ID_SEDOL1	ID002
/bbgid	/<identifier>	ID_BB_GLOBAL	ID135
/ticker	/<identifier>	PARSEKYABLE_DES_SOURCE	DX194 and DS587

Market list requests with the secids subservice name are always IMMUTABLE, which means that the returned list of instruments does not receive update messages and must be re- requested to discover any new pricing sources that emerge after the initial request. Listed below are the Market List Requests with the Secids Subservice Name:

Key Field	Format	Result
Bloomberg Unique ID	//blp/mktlist/secids/buid/ uniqueid	All instrument IDs for the given buid
	//blp/mktlist/secids/buid/EQ0010080100001000	
Bloomberg Symbol	//blp/mktlist/secids/bsym/ symbol	All instrument IDs for the given bsym
	//blp/mktlist/secids/bsym/VOD	
SEDOL	//blp/mktlist/secids/sedol/ sedol	All instrument IDs for the given SEDOL
	//blp/mktlist/secids/sedol/2005973	
CUSIP	//blp/mktlist/secids/cusip/ cusip	All instrument IDs for the given CUSIP
	//blp/mktlist/secids/cusip/459200101	
ISIN	//blp/mktlist/secids/isin/ isin	All instrument IDs for the given ISIN
	//blp/mktlist/secids/isin/US4592001014	
Parsekeyable	//blp/mktlist/secids/bpkbl/ parsekeyable	All instrument IDs for the given parsekeyable
	//blp/mktlist/secids/bpkbl/UKX Index	

Listed below are the Market List Requests with the Secids Subservice Name:

Key Field	Format	Result
Message Scraping (MSG1)	//blp/mktlist/secids/bsym/ MSGSCR P	The list of MSG1 instruments.
	//blp/mktlist/secids/bsym/ MSGSCR P	
Bloomberg Global ID	//blp/mktlist/secids/bbgid/ globalid	All instrument IDs for the given bbgid
	//blp/mktlist/secids/bbgid/BBG000BLNNH6	
Bloomberg Ticker	//blp/mktlist/secids/ticker/ symbol	All instrument IDs for the given ticker
	//blp/mktlist/secids/ticker/IBM US Equity	

A security-based secids request can also be modified to limit the source using the 'source' parameter. This table demonstrates such an instrument with and without the "source" parameter. Listed below are the Market List Requests with the Secids Subservice Name:

Subscription String	Returns
//blp/mktlist/secids/cusip/459200101	This example returns all IDs for the given CUSIP.
//blp/mktlist/secids/cusip/459200101;source=US	This example returns all IDs for the given CUSIP, but limited to source US.

The following code snippet demonstrates how to request static market list snapshot data and assumes that a session already exists and that the `"/blp/mktlist"` service has been successfully opened.

```
const char *security = "//blp/mktlist/secids/cusip/459200101;source=US";
Service mktListService = session.getService("/blp/mktlist");
Request request =
mktListService.createRequest("SnapshotRequest");
request.set("security", security);
```

6.2.8. Data Response For "secids" Snapshot Request

The following data response is associated with the snapshot request code snippet.

```
SnapshotRequest = { security = //blp/mktlist/secids/cusip/
459200101;source=US }

LIST_ID =
//blp/mktlist/secids/cusip/459200101;source=US EID =
35009
LIST_LISTTYPE = Security IDs
LIST_INSERT_ENTRIES
    ID_BB_SEC_NUM_SRC =
    399432473346 FEED_SOURCE = US
    ID_BB_SEC_NUM_DES = IBM
    ID_BB_UNIQUE =
    EQ0010080100001000
    SECURITY_TYP2 = Equity
LIST_ORDERED = NOTORDERED
LIST_MUTABLE = IMMUTABLE
```

In their application, users will handle the data response the same way, initially, as they would for any static request. This is accomplished by checking the event type of the incoming message. If its event type is `PARTIAL_RESPONSE`, then that indicates that there is at least one more message to be received to fulfill that request. Users will continue reading the incoming messages until they receive a `RESPONSE` event type, which indicates that the request has been fully served.

☞ For additional information on refer to the *"Reference Services and Schemas Guide"*.

Here is a sample event handler written in C++. It was extracted from the `"MarketListSnapshotExample"` example found in the B-PIPE C++ API SDK, and is the event handler that is responsible for displaying the above output to a console window.

```
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;
                }
            }
    }
}
```

```

// return true if processing is completed, false otherwise
void processResponseEvent(Event event)
{
    MessageIterator msgIter(event);
    while (msgIter.next()) {
        Message msg =
            msgIter.message(); Element
            responseCode;
        if ((msg.asElement().getElement(&responseCode, "responseCode") == 0) &&
            !responseCode.isNull())
        {
            int resultCode =
                responseCode.getElementAsInt32("resultCode"); if (resultCode
                > 0)
            {
                std::string message =
                    responseCode.getElementAsString("resultCode"); std::string
                    sourceId = responseCode.getElementAsString("sourceId");
                std::cout << "Request Failed: " << message << std::endl;
                std::cout << "Source ID: " << sourceId << std::endl;
                std::cout << "Result Code: " << resultCode <<
                    std::endl; continue;
            }
        }

        Element snapshot =
            msg.getElement("snapshot"); size_t
            numElements = snapshot.numElements(); for
            (size_t i = 0; i < numElements; ++i)
        {
            const Element dataItem = snapshot.getElement(i);
            // Checking if the data item is Bulk data
            item if (dataItem.isArray()){
                processBulkData(dataItem);
            }else{
                std::cout << "\t" << dataItem.name() << " = " <<
                    dataItem.getValueAsString() << std::endl;
            }
        }
    }
}

```

If users examine the response from the example market list request, which is `//blp/mktlist/secids/cusip/459200101;source=US`, they will find that the data is all returned in a single message, which means that the message will have an event type of "RESPONSE". Within that block of code, there is a call to `processResponseEvent()`. It is here that a check is made first for the `responseCode` element. To understand the reason for checking for this element, users will first need to understand the structure of the schema for the `//blp/mktlist` service. Displayed below is a screenshot capturing the sub-elements of the `SnapshotRequest/Responses` node.

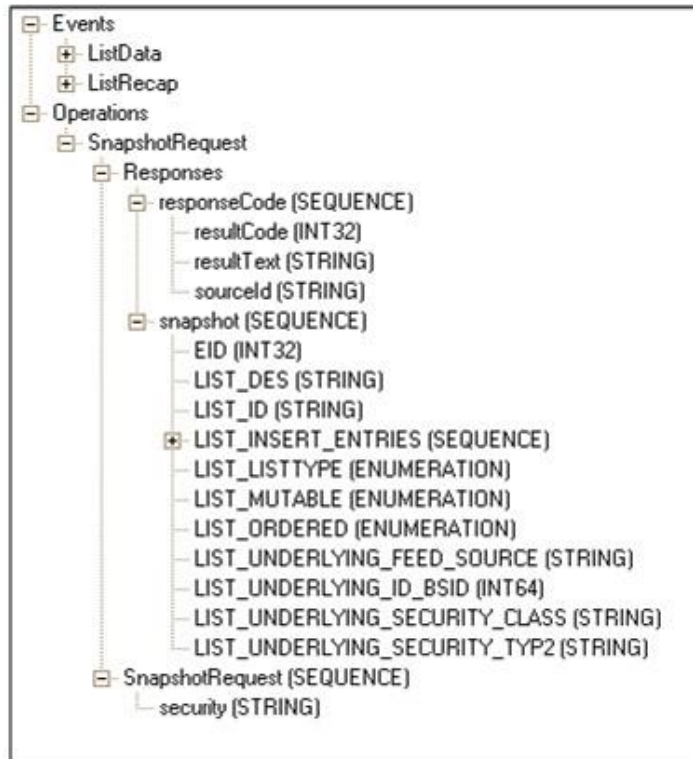


Figure 6: SnapshotRequest/Responses node

If the responseCode is found in the message, then users will check to see if the resultCode is greater than zero. If it is, then this is an indication that there was a problem with the request and that this message contains an error. The details of the error will be provided by the message's resultCode, resultText and sourceId values.

If the resultCode equals zero, then data can be expected to be contained within the message. In this case, the snapshot element of the message will be retrieved. It will be seen that in the above processResponseEvent() handler that the number of elements contained in the snapshot are determined by a call to numElements() and then each of those elements are then read into a dataItem variable, of type Element, one at a time. Users can check to see if the dataItem is an array by calling its isArray() function. If it returns true, then it is an array containing one, or more, items and must be processed differently than if containing a single item.

Users will see in the schema screenshot that there are a total of ten possible single field elements and one array element in a snapshot. The array element is indicated by the SEQUENCE type. In this case, the resultCode is zero (i.e. no errors) and there are 6 elements contained in the snapshot element. The first 3 of them are single field elements (e.g. LIST_ID, EID, LIST_LISTTYPE), which means that isArray() returns false for each of them. However, the 4th element, LIST_INSERT_ENTRIES, is an array (a.k.a. SEQUENCE type). This element is then processed in the processBulkData() function. The remaining two elements (LIST_ORDERED and LIST_MUTABLE) are also single field elements.

6.3. Source Reference Service (//blp/srcref)

6.3.1. Overview

The Source Reference and Tick Size subscription service (aka //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.

Listed below are the Source Reference String Definitions:

Source Reference Name	Description
<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)

There are currently four subservices that can be used in a user's subscription string. Listed below are the 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:

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

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

6.3.2. Important BPOD Upgrade Notes:

1. B-PIPE breaks down the subscriptions into a more granular format. With BPOD, users would have subscribed to "//blp/mktref/scref/eid/<eid>" to obtain all source references for that EID, which included the broker codes, trade condition codes, quote condition codes, market depth condition codes, period suspense codes, security suspense codes and ticksizes. Now, by using B-PIPE, users can break down these source references into four main subscriptions:
 "//blp/scref/brokercodes/eid/<eid>", "//blp/scref/conditioncodes/eid/<eid>", "//blp/scref/tradingstatuses/eid/<eid>" and "//blp/scref/ticksizes/eid/<eid>".
2. B-PIPE has introduced filters for some of its subservices to allow users to subscribe to the data they are most interested in.
3. With B-PIPE, a description message is returned for each subservice's sources.
4. With B-PIPE, Bloomberg now offers intraday updating for tick size changes.
5. If users are looking for the sources on contributor EIDs (or any EID), they should subscribe to //blp/scref for any of the subservices (i.e. /ticksizes, /brokercode, etc) and the list of descriptions for that source will be included even if the subservice doesn't apply. For example, "//blp/scref/ticksizes/eid/14240" will return the sources for 14240, but there will not be any ticksizes information.

6.3.3. Code Example

A SourceRefSubscriptionExample is found in the B-PIPE SDK for C++, Java and .NET. This C++ example demonstrates how to make a simple Source Reference subscription for the

condition codes associated with EID 14003. Displayed is the C++ code snippet - subscribing for a list of condition codes for EID 14003.

```
const char *list = "//blp/srcref/conditioncodes/eid/14003";
SubscriptionList subscriptions;
subscriptions.add(list, CorrelationId((char *)security));
session.subscribe (subscriptions);
```

6.3.4. Response Overview

The Source Reference response will be a series of SUBSCRIPTION_DATA events, which users will be familiar with if they have developed Bloomberg API applications using any of the other streaming services, such as //blp/mktdata, //blp/mktlist or //blp/mktdepthdata.

All SUBSCRIPTION_DATA event messages will be of message type SourceReferenceUpdates and will contain a SOURCE_REF_EVENT_TYPE_RT (event type), SOURCE_REF_EVENT_SUBTYPE_RT (event sub-type) and EID field (int32), along with an array of event type field items applicable to the subservice users are subscribing to.

The table below^{bookmark148} lists the possible enumeration values for the event type and event sub-type fields:

Name	Description	Values
SOURCE_REF_EVENT_TYPE_RT	This specifies the event type.	Possible enumeration values: DESCRIPTION_BROKER_CODE TRADE_CONDITION_CODE QUOTE_CONDITION_CODE MKTDEPTH_CONDITION_CODE TRADING_PERIOD TRADING_STATUS TICK_SIZE_TABLE
SOURCE_REF_EVENT_SUBTYPE_RT	This specifies the event sub-type	Possible enumeration values: INITPAINT - Initial Paint REFRESH - Daily Refresh ^a UPDATE - Intraday Update

a. Refreshes are performed daily at approximately 6pm (Eastern Standard Time).

The subservice name included in the user's subscription will dictate which event type (SOURCE_REF_EVENT_TYPE_RT) field items will be returned as initial snapshot (INITPAINT) and refresh sub-type messages. The table below will assist users in determining which SOURCE_REF_EVENT_TYPE_RT field types to expect based on the subservice in their subscription.

6.3.5. Response Event Types by Subservice

The table below lists the entire initial snapshot and refresh (i.e., INITPAINT and REFRESH, respectively) event type fields users should expect to receive for the subservice they are subscribing to.

Subservice Name	Response Event Types
/brokercodes	DESCRIPTION + BROKER_CODE
/conditioncodes	DESCRIPTION + TRADE_COND_CODE + QUOTE_COND_CODE + MKTDEPTH_COND_CODE
/tradingstatuses	DESCRIPTION + TRADING_PERIOD + TRADING_STATUS
/ticksizes ^a	DESCRIPTION + TICK_SIZE_TABLE

a. All subservices will return INITPAINT and REFRESH event messages. However, /ticksizes will also return UPDATE event messages."

For a breakdown of each message returned for the subservice, please see the table below.

6.3.6. Breakdown of Event Type Fields

The table below describes the breakdown of each event type's field array. Each name given to the field array is the pluralized form of the aforementioned event type value (e.g., The DESCRIPTION event type value (as found in Table above) will have an associated field array name of DESCRIPTIONS).

Field Name	Type	Contents
DESCRIPTIONS	SourceReferenceDescriptions	Contains the feed EID and feed source, along with a list of DESCRIPTION entries containing each item's expanded name of the data contributor or exchange and local source of the composite source for lookup to condition code and broker.
BROKER_CODES	SourceReferenceBrokerCodes	Contains the feed EID and feed source, along with a list of BROKER_CODE entries containing each item's Bloomberg mnemonic and associated name.
TRADE_COND_CODES	SourceReferenceTradeConditionCodes	Contains the feed EID and feed source, along with a list of TRADE_COND_CODE entries containing each item's Bloomberg mnemonic(s) for special conditions on the trade, condition code, trade category, short name for the sale condition, ESMA transaction code and more.
QUOTE_COND_CODES	SourceReferenceQuoteConditionCodes	Contains the feed EID and feed source, along with a list of QUOTE_COND_CODE entries containing each item's quote condition mnemonic, Bloomberg condition code, quote condition short name and Provider assigned condition code mnemonic(s).
MKTDEPTH_COND_CODES	SourceReferenceMarketDepthConditionCodes	Contains the feed EID and feed source, along with a list of MKTDEPTH_COND_CODE entries containing each item's Bloomberg mnemonic, for the condition, short name for the condition and Provider assigned condition code mnemonic(s).

Field Name	Type	Contents
TRADING_PERIODS	SourceReferenceTradingPeriods	Contains the feed EID and feed source, along with a list of TRADING_PERIOD entries containing each item's Bloomberg assigned mnemonic for the current trading period of a security, Bloomberg's short name for the current trading period of the security, and Bloomberg's assigned simplified status mnemonic for the current market status of a security.
TRADING_STATUSES	SourceReferenceTradingStatuses	Contains the feed EID and feed source, along with a list of TRADING_PERIOD entries containing each item's Bloomberg assigned mnemonic for the current trading status of a security, Bloomberg's short name for the market status on a source, and Bloomberg's assigned simplified status mnemonic for the current market status of a security.
TICK_SIZE_TABLES	TickSizeTable	Contains the feed EID, feed source, table field name, table identifier, percent field name, table type and frequency at which the tick size can change, along with a list of TICK_SIZE_TABLE_ROW entries containing each item's type of tick size value, lower/upper bounds value, and tick size value used for the range.

6.3.7. Handling Multiple Messages (a.k.a. Fragments)

- It is noticed that initial paint messages can be split into one or more smaller messages in the case where the returned data is too large to fit into a single message. It will be up to the user to handle this in their application.
- Users will achieve this by checking the Fragment type of any SUBSCRIPTION_DATA event SourceReferenceUpdates message. The Fragment enum is used to indicate whether a message is a fragmented message or not and what position it occurs within the chain of split fragmented messages. If the SourceReferenceUpdates is split into two parts, then the first message will have a Fragment type value of FRAGMENT_START and a last message of FRAGMENT_END. If the SourceReferenceUpdates is split into more than 2 parts, all middle Fragments will be of type FRAGMENT_INTERMEDIATE. Displayed below are the Fragment Type Enumerators:

Message::Fragment Type Enumerators	
FRAGMENT_NONE	Message is not fragmented
FRAGMENT_START	The first fragmented message
FRAGMENT_INTERMEDIATE	Intermediate fragmented messages
FRAGMENT_END	The last fragmented message

6.3.8. Data Response for Subscription

Here is sample output for a Source Reference subscription to "//blp/srcref/ticksizes/eid/ 14014":

```
*****
*   INITIAL SNAPSHOT
***** SourceReferenceUpdates
= {
    SOURCE_REF_EVENT_TYPE_RT =
    DESCRIPTION
    SOURCE_REF_EVENT_SUBTYPE_RT =
    INITPAINT EID = 35009
    DESCRIPTIONS[] =
        DESCRIPTIONS = {
            FEED_SOURCE
            = LN
            FEED_EID =
            14014
            DESCRIPTION[
            ] =
                DESCRIPTION = {
                    FEED_SOURCE_DES_RT = London Stock Exchange Domestic
                }
        }
    -- MORE --
}

-----
SourceReferenceUpdates = {
    SOURCE_REF_EVENT_TYPE_RT =
    TICK_SIZE_TABLE
    SOURCE_REF_EVENT_SUBTYPE_RT =
    INITPAINT EID = 35009

    TICK_SIZE_TABLES[] =
        TICK_SIZE_TABLES =
        {
            FEED_SOURCE = LN
            FEED_EID = 14014
TICK_SIZE_TABLE_IDENTIFIER_RT = 2871
TICK_SIZE_TABLE_TYPE_RT = PRICE
TICK_SIZE_TABLE_UPDATE_FREQ_RT = DAILY
TICK_SIZE_TABLE_FIELD_NAME_RT = LAST_TRADE

            TICK_SIZE_TABLE_ROW[] = TICK_SIZE_TABLE_ROW = {
                TICK_SIZE_TABLE_PRICE_TYPE_RT = ABSOLUTE
                TICK_SIZE_TBL_BAND_TICK_SIZE_RT = 0.050000
                TICK_SIZE_TBL_BAND_TICK_SIZE_RT = 0.000000
                TICK_SIZE_TBL_BAND_TICK_SIZE_RT = 10000000000.000000
            }
        }
    -- MORE --
}
```

```

*****
* DAILY REFRESH
*****
*** SourceReferenceUpdates = {
    SOURCE_REF_EVENT_TYPE_RT =
    DESCRIPTION
    SOURCE_REF_EVENT_SUBTYPE_RT =
    REFRESH EID = 35009
    DESCRIPTIONS[] =
        DESCRIPTIONS =
        {
            FEED_SOURCE =
            LN FEED_EID =
            14014
            DESCRIPTION[]
            =
                DESCRIPTION = {
                    FEED_SOURCE_DES_RT = London Stock Exchange Domestic
                }
        }
-- MORE --
}
SourceReferenceUpdates = {
    SOURCE_REF_EVENT_TYPE_RT =
    TICK_SIZE_TABLE
    SOURCE_REF_EVENT_SUBTYPE_RT = REFRESH
    EID = 35009
    TICK_SIZE_TABLES[]
    =
        TICK_SIZE_TABLES
        =
        {
            FEED_SOURCE =
            LN FEED_EID =
            14014

TICK_SIZE_TABLE_IDENTIFIER_RT = 5977 TICK_SIZE_TABLE_TYPE_RT = PRICE

            TICK_SIZE_TABLE_ROW[] = TICK_SIZE_TABLE_ROW = {
                TICK_SIZE_TABLE_PRICE_TYPE_RT = ABSOLUTE
                TICK_SIZE_TBL_BAND_TICK_SIZE_RT = 0.050000
                TICK_SIZE_TBL_BAND_TICK_SIZE_RT = 0.000000
                TICK_SIZE_TBL_BAND_TICK_SIZE_RT = 10000000000.000000
            }
        }
-- MORE --
}
*****
* DAILY REFRESH
*****
*** SourceReferenceUpdates = {
    SOURCE_REF_EVENT_TYPE_RT =
    DESCRIPTION
    SOURCE_REF_EVENT_SUBTYPE_RT =
    REFRESH EID = 35009
    DESCRIPTIONS[] =

```

```

        DESCRIPTIONS =
        {
            FEED_SOURCE =
            LN_FEED_EID =
            14014
            DESCRIPTION[]
            =
                DESCRIPTION = {
                    FEED_SOURCE_DES_RT = London Stock Exchange Domestic
                }
        }
    -- MORE --
}
SourceReferenceUpdates = {
    SOURCE_REF_EVENT_TYPE_RT =
    TICK_SIZE_TABLE
    SOURCE_REF_EVENT_SUBTYPE_RT = REFRESH
    EID = 35009
    TICK_SIZE_TABLES[]
    =
        TICK_SIZE_TABLES
        =
        {
            FEED_SOURCE =
            LN_FEED_EID =
            14014
TICK_SIZE_TABLE_IDENTIFIER_RT = 5977 TICK_SIZE_TABLE_TYPE_RT = PRICE

            TICK_SIZE_TABLE_UPDATE_FREQ_RT = DAILY
            TICK_SIZE_TABLE_FIELD_NAME_RT = LAST_TRADE
            TICK_SIZE_TABLE_ROW[] =
            TICK_SIZE_TABLE_ROW =
            {
                TICK_SIZE_TABLE_PRICE_TYPE_RT = ABSOLUTE
            }
            TICK_SIZE_TABLE_ROW = {
                TICK_SIZE_TABLE_PRICE_TYPE_RT =
                ABSOLUTE
            }
        }
    -- MORE --
}

```

```

*****
*   TICKSIZE INTRADAY UPDATE
*****
SourceReferenceUpdates = {
    SOURCE_REF_EVENT_TYPE_RT = TICK_SIZE_TABLE
    SOURCE_REF_EVENT_SUBTYPE_RT = UPDATE
    EID = 35009
    TICK_SIZE_TABLES[] =
        TICK_SIZE_TABLES =
            {
                FEED_SOURCE
                = LN FEED_EID
                = 14014

                TICK_SIZE_TABLE_IDENTIFIER_RT = 5995
                TICK_SIZE_TABLE_TYPE_RT = PRICE
                TICK_SIZE_TABLE_UPDATE_FREQ_RT = DAILY
                TICK_SIZE_TABLE_FIELD_NAME_RT = LAST_TRADE
                TICK_SIZE_TABLE_ROW[] =
                    TICK_SIZE_TABLE_ROW = {
                        TICK_SIZE_TABLE_PRICE_TYPE_RT = ABSOLUTE
                        TICK_SIZE_TBL_BAND_TICK_SIZE_RT = 0.300000

                        TICK_SIZE_TBL_BAND_LOWER_VAL_RT = 0.250000

                        TICK_SIZE_TBL_BAND_UPPER_VAL_RT = 100000000.000000
                    }
            }
}

-- MORE --
}

```

In the above sample output, a subscription containing the subservice "/ticksizes" was made, which means that a user can expect to receive "INITPAINT" and "REFRESH" event types (i.e. SOURCE_REF_EVENT_TYPE_RT) messages containing "DESCRIPTION" and "TICK_SIZE_TABLE" event sub-types (i.e. SOURCE_REF_EVENT_SUBTYPE_RT). In addition to the aforementioned messages, which are standard for all of the subservice requests, they will also receive "UPDATE" event type messages, which are unique to the / ticksizes subservice. However, there will not be an UPDATE "DESCRIPTION" message sent.

Taking a look at the sample output above, users will notice that every SourceReferenceUpdates message contains the standard event type, sub-type and EID single-value fields, along with an array of fields applicable for that event type. For instance, in the message containing the event type of "TICK_SIZE_TABLE" they will find an array of "TICK_SIZE_TABLES" fields.

6.4. Message Scraping Service (//blp/msgscrape)

6.4.1. Introduction

B-PIPE offers a real-time feed of MSG/IB scraped data. This feed will contain the complete set of mined quotes received on the MSG Minding PCS Group.

The real-time feed will include price quotes that are both verified and not verified. These are equivalent to GREEN and WHITE quotes on **IMGR<GO>**. Clients will need to be enabled for a MSG1

account, real-time feeds for these accounts, and nominate a four-letter mnemonic to represent the feed (PCS). An EID will be created to represent this account (e.g. 44321).

Message Scraping Service: Subscriptions

These MSG1 accounts are split into two flavors; Generated Data Accounts (for the Sell Side) and Received Data Accounts (for the Buy Side).

Generated Data Accounts (Sell Side):

These accounts contain pricing created by the sell side desks of the firm consuming the feed. Users and Applications can be entitled for the account's EID in **EMRS<GO>** and a subscription to the feed created via the following:

```
subscriptions.add(new Subscription("MSGSCRAP <PCS> Curncy", correlation_id);
```

Received Data Accounts (Buy Side):

These accounts contain pricing generated by counterparties (e.g. brokers from the sell side) at other firms and sent to the buy side of the account owners. This data is limited to consumption by users who are members of the account on the **SXBM<GO>** function on the Bloomberg Professional Service. The data is also permitted for use with Derived Data Applications provided that the resultant data does not enable reverse engineering to the raw prices.

If data is requested via user-mode then it will subscribe a user to the correct stream of data automatically based on their account settings. The subscription for this would be as follows:

```
subscriptions.add(new Subscription("MSGSCRAP <PCS> Curncy", correlation_id);
```

Note the usage of the explicit pricing source MSG1 for buy side accounts as opposed to the user defined <PCS> for the Sell Side users.

An Application can potentially access multiple MSG1 accounts, and so specific accounts need to be explicitly requested by use of the EID.

```
subscriptions.add(new Subscription("MSGSCRAP MSG1 Curncy", EID="44321", correlation_id));
```

6.4.2. Heartbeat:

In addition to the message scraped data, the service is also able to supply a heartbeat instrument that users and applications could subscribe to. This ticks every 5 seconds and the intent was that clients have a method of determining whether a feed remains active.

```
subscriptions.add(new Subscription("MSGHBEAT Index?fields=TIME", correlation_id));
```

This can be subscribed to as follows:

```
TRADE_SIZE_ALL_SESSIONS_RT = 100
EID = 14005
IS_DELAYED_STREAM = false
```

7. New Book Discovery Service

The market depth data service (**mktdepthdata** - `//blp/mktdepthdata`) provides EPS customers with market depth data streams which they can monitor. In order to make a subscription, the client needs to know what books and views are available and whether he is entitled to view these streams or not. Currently there is no means provided for EPS customers to discover this information.

A new service endpoint for the **mktdepthdata** service is proposed that will satisfy this requirement. This service endpoint will be the market data book service (**BookListService**). The aim of this service endpoint is to provide the parameters required for a client to build the UTS string in order to make a call to the market depth data service, as well as an indication as to whether he is entitled to this stream and (possibly) other meta-data.

The **mktbooksvc** task which will resolve requests for the **BookListService** service will be a Solaris/AIX only task since access to specific BBG functionality on these platforms is required, which cannot be satisfied on Linux.

7.1. Requirements

7.1.1. Introduction

The service endpoint will be a request/response endpoint (known as a 'static' service in BBG API terminology). The current **mktdepthdata** service endpoint provides subscription ('realtime' in BBG API terminology) access. A service may be configured with both realtime and static endpoints. The access will be via the **mktdepthdata** service's **BookListService** endpoint.

7.1.2. Request

The request message will contain a list of parameters that we require the book information for. The request will:

- support multiple queries in one request;
- **Security:** query on a security. This allows the user to check what entitlements are required and what books are available for a security. All formats supported by the service may be used (ticker, ISIN, ...). The format is 'ticker/VOD LN Equity' (ticker), 'isin/GB00BH4HKS39 LN Equity' (ISIN), etc. The format is the same as for the subscription service.

Mixed queries (EID and Security) will not be allowed, but separate queries for EID and Security will be allowed in the same request.

The structure of the request will be:

- Request
 - Query[...]
 - Query type

- Query data

For practical purposes the number of the queries in the request MUST be limited. This is best handled by the service, rather than being defined in the service XML schema. All queries past the limit will not be processed and an error returned in their reply.

In addition, the service will have access to the calling user context in order to ascertain matching entitlements.

7.1.3. Response

There will be only one response message. It will contain one reply for each request query. In the case of an error, this will be flagged in the reply with the appropriate error message. This will obviate the necessity for the client to handle out of band error messages. The query parameter will also be included in the reply so the user does not have to assume any order of replies not do any matching of request to response. This will simplify asynchronous processing of the responses. Although the replies will probably be in the same order as the queries, we do not want to have to make this a requirement.

The reply content will differ depending on the query type. The structure of the two reply types will be the same, though:

- query by EID has been deprecated; and
- **Security:** all the available books for this security will be returned. The default book, and user entitlement (EID) required for these books will also be returned. In addition, each book will be flagged as to whether the caller's entitlements match the EID for that book.

The structure of the response will be:

- Reply[...]
 - Query type
 - Query data
 - Exchange[...]
 - exchange indicator (some sort of exchange identifier)
 - Asset[...]
 - asset type indicator
 - Type [...]
 - type indicator (MBO/TOP/MBL)
 - Asset subtype[...]
 - subtype indicator (options, etc...)
 - Book
 - exchange segment (if relevant)
 - exchange semi-configured flag (in case of exchanges where book availability is security dependent)
 - EID
 - EID match flag
 - View Mnemonic - the **mktdepthdata** UTS parameter

- Default book flag
- additional field(s) (if required)

7.1.4. System configuration

Since the proposed service is a request/response service, no continual connection to the service is required. In addition, the service is stateless.

With this in mind, there is no requirement for any failover. If an instance becomes unavailable, the API infrastructure will simply route further requests to another instance of the service.

For failure and load balancing requirements, multiple instances of the service across multiple servers will be required. It is not currently envisaged to run more than one instance per server.

The service will only be deployed on Solaris and AIX machines.

7.1.5. Performance metrics

There are currently no metrics as to required response times, nor loading in transactions per second.

7.1.6. Service schema

The service schema for the `mktbooksvc` task will be an extension of the `mktdepthdata` schema. Both services will share the same schema. The `mktdepthdata` service task will resolve the realtime service ID endpoints and the `mktbookdata` task will resolve the static service ID endpoints via the `BookListService` service.

The following will be added to the `mktdepthdata` schema:

<ServiceDefinition ... >

<service>

<!-- Book List Service request -->

<sequenceType name="BookListRequest">

<description>Request for the list of available books using EID or Parsekeyable.</description>

<element name="queries" type="BookListQueryType" minOccurs="1" maxOccurs="unbounded">

<description>Array of queries for book lists.</description>

</element>

</sequenceType>

<!-- Book List Service response -->

<sequenceType name="BookListResponse">

<description>Response for the list of available books.</description>

<element name="responses" type="BookListResponseType" minOccurs="1" maxOccurs="unbounded">

<description>Array of responses for the book list queries - one for each query in the request.</description>

</element>

</sequenceType>

<!-- Book List Service request query type -->

```
<sequenceType name="BookListQueryType">
  <description>Request for the list of available books.</description>
  <element name="queryType" type="BookListQueryTypeEnum">
    <description>The type of query the query contains.</description>
  </element>
  <element name="query" type="String" minOccurs="1" maxOccurs="1">
    <description>The query value.</description>
  </element>
</sequenceType>
```

<!-- The list of Book List Service query type enumerations -->

```
<enumerationType name="BookListQueryTypeEnum" type="Int32" >
  <description>The request query type.</description>
  <enumerator name="Ticker">
    <description>Query parameter is the ticker.</description>
    <value><Int32>0</Int32></value>
  </enumerator>
</enumerationType>
```

<!-- Book List Service response type -->

```
<sequenceType name="BookListResponseType">
  <element name="query" type="BookListQueryType">
    <description>The query from the request.</description>
  </element>
  <element name="queryStatus" type="Boolean">
    <description>Status of the query: true is OK, false is an error (see errorMsg for details).</description>
  </element>
  <element name="errorMsg" type="String">
    <description>A string representation of the error for this query (if any).</description>
  </element>
  <element name="ticker" type="String" minOccurs="0" maxOccurs="1">
    <description>Parsekeyable.</description>
  </element>
  <element name="bookList" type="BookListInfo" minOccurs="0" maxOccurs="unbounded">
    <description>Exchange data structures. May be empty in the case of an error.</description>
  </element>
</sequenceType>
```

<!-- Book List Service exchange type -->

```
<sequenceType name="BookListInfo">
  <description>Book List service type to hold books grouped by exchange.</description>
  <element name="exchangeName" type="String">
    <description>The unique name of this exchange</description>
  </element>
  <element name="bookListAllAssets" type="BookListPerAssetInfo" minOccurs="1" maxOccurs="unbounded">
    <description>Asset data structures.</description>
  </element>
</sequenceType>
```

<!-- Book List Service asset type -->

```
<sequenceType name="BookListPerAssetInfo">
```

```
<description>Book List Service type to hold information for each asset.</description>
<element name="assetType" type="BookListAssetPerInfoEnum">
<description>The type for this asset.</description>
</element>
<element name="booksPerAsset" type="BookListPerRequestTypeInfo" minOccurs="1"
maxOccurs="unbounded">
<description>List of functions (MBO/TOP/MBL/...) for this asset.</description>
</element>
</sequenceType>
```

<!-- Book List asset (yellow key) types -->

```
<enumerationType name="BookListAssetPerInfoEnum" type="Int32">
<description>The asset types. Must match Bloomberg internal list (products.h).</description>
<enumerator name="CMDT">
<description>Asset commodity type.</description>
<value><Int32>1</Int32></value>
</enumerator>
<enumerator name="EQTY">
<description>Asset equity type.</description>
<value><Int32>2</Int32></value>
</enumerator>
<enumerator name="MUNI">
<description>Asset municipal type.</description>
<value><Int32>3</Int32></value>
</enumerator>
<enumerator name="PRFD">
<description>Asset preferred type.</description>
<value><Int32>4</Int32></value>
</enumerator>
<enumerator name="CLNT">
<description>Asset clnt type.</description>
<value><Int32>5</Int32></value>
</enumerator>
<enumerator name="MMKT">
<description>Asset money market type.</description>
<value><Int32>6</Int32></value>
</enumerator>
<enumerator name="GOVT">
<description>Asset government type.</description>
<value><Int32>7</Int32></value>
</enumerator>
<enumerator name="CORP">
<description>Asset corporate type.</description>
<value><Int32>8</Int32></value>
</enumerator>
<enumerator name="INDX">
<description>Asset index type.</description>
<value><Int32>9</Int32></value>
</enumerator>
<enumerator name="CURR">
<description>Asset currency type.</description>
<value><Int32>10</Int32></value>
</enumerator>
<enumerator name="MTGE">
<description>Asset mortgage type.</description>
```

```
<value><Int32>11</Int32></value>
</enumerator>
<enumerator name="ACTN">
<description>Asset auction type.</description>
<value><Int32>12</Int32></value>
</enumerator>
</enumerationType>
```

<!-- Book List Service function type -->

```
<sequenceType name="BookListPerRequestTypeInfo">
<description>Book List Service type to hold information for each function.</description>
<element name="requestType" type="BookListPerRequestTypeInfoEnum">
<description>The type for this function.</description>
</element>
<element name="bookListAllSubtypes" type="BookListAssetSubtypeInfo" minOccurs="1"
maxOccurs="unbounded">
<description>List of asset subtypes for this function</description>
</element>
</sequenceType>
```

<!-- Book List Service enumeration of function types -->

```
<enumerationType name="BookListPerRequestTypeInfoEnum" type="Int32">
<description>The function types. These map to internal MdmadbFunc in
mdmadb_utils.h.</description>
<enumerator name="MBO">
<description>MBO(BBO) function type.</description>
<value><Int32>0</Int32></value>
</enumerator>
<enumerator name="MBL">
<description>MBL(MQ) function type.</description>
<value><Int32>1</Int32></value>
</enumerator>
<enumerator name="TOP">
<description>TOP function type.</description>
<value><Int32>2</Int32></value>
</enumerator>
<enumerator name="MMQ">
<description>MMQ function type.</description>
<value><Int32>3</Int32></value>
</enumerator>
</enumerationType>
```

<!-- Book List Service asset subtype type -->

```
<sequenceType name="BookListAssetSubtypeInfo">
<description>Book List Service type to hold information for each asset subtype.</description>
<element name="subtype" type="Int32">
<description>The value for this asset subtype.</description>
</element>
<element name="books" type="BookListBookInfo" minOccurs="1" maxOccurs="unbounded">
<description>List of books for this asset subtype.</description>
</element>
</sequenceType>
```

<!-- Book List Service book type -->

```
<sequenceType name="BookListBookInfo">
```

```
<description>Book List Service type to hold the book information.</description>
<element name="permissionLevel" type="Int32">
<description>The book permission level</description>
</element>
<element name="exchangeSegment" type="String">
<description>The exchange segment (if relevant)</description>
</element>
<element name="defaultBook" type="Boolean">
<description>Flag whether this is the default book or not.</description>
</element>

<element name="eid" type="Int32">
<description>The EID associated with this book.</description>
</element>
<element name="eidMatch" type="Boolean">
<description>Flag whether the caller's permissions contain the required EID.</description>
</element>
<element name="viewMnemonic" type="String">
<description>The mktdepthdata view parameter value.</description>
</element>
<element name="bookValues" type="BookListBookValueType" minOccurs="0"
maxOccurs="unbounded">
<description>List of field name/value pairs for this book.</description>
</element>
</sequenceType>

<!-- Book List Service book value type -->
<sequenceType name="BookListBookValueType">
<element name="name" type="String">
<description>Name of this element, e.g. ' Levels Maintained'.</description>
</element>
<element name="value" type="String">
<description>Value of this element, e.g. '200'.</description>
</element>
</sequenceType>
```