SYBASE®

Developers Guide

Sybase RAP - The Trading Edition

DOCUMENT ID: DC00794-01-0100-01

LAST REVISED: March 2008

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

About This Book

Audience

Sybase RAP - The Trading Edition Developers Guide is intended for developers who are creating custom feed handlers and statistics monitoring for Sybase RAP.

How to use this book

Before using the information in this book to write applications that interface between market data feeds and the Sybase RAP components, refer to the *Sybase RAP - The Trading Edition Release Bulletin* for any last minute information regarding this product.

Related documents

Refer to the following documents for more information:

- *Sybase RAP - The Trading Edition Release Bulletin*
- *Sybase RAP - The Trading Edition Installation and Configuration Guide*
- *Sybae RAP - The Trading Edition Users Guide*
- *Sybase RAP - The Trading Edition Operations Console Users Guide*
- Sybase IQ 12.7 product documentation
- Adaptive Server® Enterprise 15.0 product documentation
- OpenSwitch™ 15.1 product documentation
- PowerDesigner® 12.5 product documentation
- Open Client™ 15.0 product documentation
- White paper titled Using Sybase NonStopIQ and EMC CLARiiON for Backup/Restore, High Availability, and Disaster Recovery at http://www.sybase.com/detail?id=1054761
- White paper titled Time Series in finance: the array database approach at http://cs.nyu.edu/shasha/papers/jagtalk.html
- White paper titled FinTime --- a financial time series benchmark at http://www.cs.nyu.edu/cs/faculty/shasha/fintime.html

---

Note This product includes software developed by The Apache Software Foundation at http://www.apache.org/.

Other sources of information

Use the Sybase Getting Started CD, the Sybase Infocenter Web site, and the Sybase Product Manuals Web site to learn more about your product:

- The Getting Started CD contains the release bulletin, installation and configuration guide, administration guide, and users guide in PDF format. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.
- The Sybase Infocenter Web site is an online version of the product manuals that you can access using a standard Web browser.

  To access the Infocenter Web site, go to Sybooks Online Help at http://infocenter.sybase.com/help/index.jsp
- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

  To access the Sybase Product Manuals Web site, go to Product Manuals at http://www.sybase.com/support/manuals/.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ **Finding the latest information on product certifications**

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Click Certification Report.
3. In the Certification Report filter select a product, platform, and timeframe and then click Go.
4. Click a Certification Report title to display the report.

❖ **Finding the latest information on component certifications**

1. Point your Web browser to Availability and Certification Reports at http://certification.sybase.com/.
2. Either select the product family and product under Search by Base Product; or select the platform and product under Search by Platform.
3. Select Search to display the availability and certification report for the selection.

❖ **Creating a personalized view of the Sybase Web site (including support pages)**

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

1 Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.

2 Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

❖ **Finding the latest information on EBFs and software maintenance**

1 Point your Web browser to the Sybase Support Page at http://www.sybase.com/support.

2 Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.

3 Select a product.

4 Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the "Technical Support Contact" role to your MySybase profile.

5 Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Accessibility features

This document is available in an HTML version that is specialized for accessibility. You can navigate the HTML with an adaptive technology such as a screen reader, or view it with a screen enlarger.

Sybase RAP documentation complies with U.S. government Section 508 Accessibility requirements. Documents that comply with Section 508 generally also meet non-U.S. accessibility guidelines, such as the World Wide Web Consortium (W3C) guidelines for Web sites.

For information about accessibility support in the Sybase IQ plug-in for Sybase Central, see “Using accessibility features” in Chapter 1, “Introducing Sybase IQ” in *Introduction to Sybase IQ*. The online help for Sybase IQ, which you can navigate using a screen reader, also describes accessibility features, including Sybase Central keyboard shortcuts.

Note You might need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce ALL UPPERCASE TEXT as initials, and MixedCase Text as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool.

For information about how Sybase supports accessibility, see Sybase Accessibility at http://www.sybase.com/accessibility. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

Note If you need to contact Sybase regarding this product, use the internal version Sybase RAP 1.0 to identify this release.

CHAPTER 1

Platform Architecture

About this Chapter

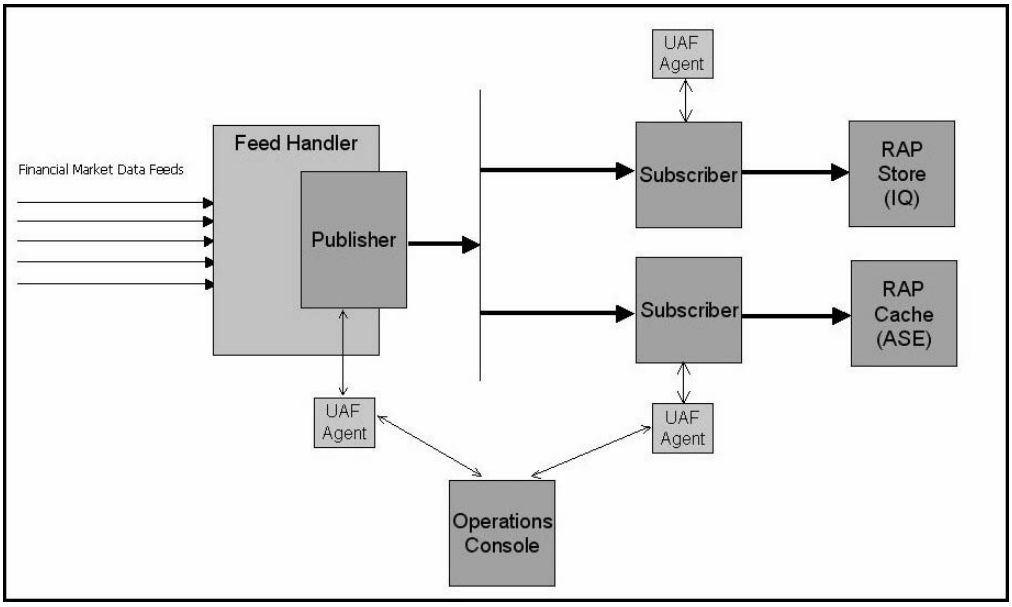
This chapter provides an overview of the software tools that can be used to write applications that interface between market data feeds and Sybase RAP components.

Contents

Overview

Sybase RAP writes in-bound market data to an in-memory cache database and a historical data store. The cache database is an Adaptive Server Enterprise (ASE) database; the historical data store is a Sybase IQ database. The platform architecture of Sybase RAP includes Feed Handlers, Publishers, and Subscribers.

Unified Agent Framework (UAF) provides a common set of discovery services, and host agent plug-ins that manage server resources or perform various operations. Agent plug-ins can perform product-specific commands, including status, start, stop, and restart.



Feed handlers

Feed handlers direct inbound market data into the Sybase RAP environment. Feed handlers are publishers that manage connectivity and message transformation directly from exchanges, like the NYSE, or consolidated service providers, like Reuters. Although all feed handlers are currently third party, Sybase RAP provides a common API publisher that allows vendors to integrate proprietary data feeds with the Sybase RAP API handler.

Sybase RAP includes a Sybase feed handler that reads comma-delimited market data from a demo file. This component is intended to help developers become familiar with the Publisher API and demonstrate network message flow.

Publishers

Publishers format incoming market data messages, and forward those messages to a subscriber in an internal Sybase RAP Data Stream (RDS) format. A publisher consists of two layers: Publisher API and Network layer.

The Publisher API is intended for Sybase feed handlers (Demo Feed Handler and FAST Feed Handler), and developers who want to map proprietary market data into Sybase RAP. The Publisher API uses an RDS template-processing module that identifies the message type, formats the messages, and calls the Network layer. The Network Layer buffers the messages into packets, and sends the packets to across multiple data streams to a subscriber.

Subscribers

Subscriber consists of two components. The Open Subscriber component, handles all network interface issues, and queues up messages to be consumed by a data stream handler component. The data stream handler routes the messages to their final destination.

Each data stream listens for packets on one UDP broadcast channel. When a packet arrives on a channel, the subscriber checks and records the packet sequence number to determine potentially missing packets. The packet recorder has a fixed size window. If the packet recorder is full, the subscriber asks the publisher to resend the missing packet on the publisher resend channel. The packet recorder will then shift its window by one packet.

Sybase RAP includes two data stream handlers: the RAPCache Data Stream Handler (Cache DSH) and the RAPStore Data Stream Handler (Store DSH). The Cache DSH loads messages into ASE. The Store DSH loads messages into Sybase IQ. The Demo Subscriber component includes a Demo Data Stream Handler that can be configured to write or discard messages.

Operations Console

Operations Console is a Web-based console that monitors system status and performs routine administration tasks. UAF plug-ins running on each server handle communication between Operations Console and Sybase RAP components. When an agent starts on a server, the agent starts all of its plug-ins and registers with the JINI server. Operations Console queries the JINI server to learn about all agents and their plug-ins.

Communication between Operations Console and the plug-ins occurs via RMI (remote method invocation). Publisher and subscriber plug-ins communicate with publishers and subscribers components via TCP over a local socket. Each publisher and subscriber has an administration channel listening on a specific port. When Operations Console issues a command, the agent intercepts the request, and contacts the component. The component sends a response through the agent back to Operations Console.

Processing market data messages

The bulk of the messaging used within the Sybase RAP system contains market data. Market data messages are buffered within a message of type `RDS_PACKET`. An `RDS_PACKET` is used as a container for one or more market data messages.

For market data messages, Sybase RAP uses a template to encode (and decode) each message in the proper format. The message format consists of a message header, a fixed-length portion of the message, and a variable-length portion. Each fixed-length field will be packed into the fixed-length portion of the message, according to the offset for the field obtained from the template. Each variable-length field will be appended to the message after the fixed-length portion, preceded by a one-byte length.

The market data message header contains the market data message type and total length. The fixed-length portion of the message appears after the market data header. Each variable-length field will follow, preceded by a single byte which indicates field length.

Building messages

Use the Sybase RAP Publisher API to create messages to the first data processing stage. The following parameter, defined in the configuration file *publisher.xml*, affects performance.

Parameter	Description
NumMessageBuffers	A message buffer is an in memory storage area used by the publisher API to store the information for a message that is in the process of being built. One message buffer is required for each message that is being simultaneously built. Multi-threaded custom applications require this value to be set at least as large as the number of threads that are concurrently creating messages. Application designers should be aware that the definition of many message buffers can result in CPU contention, and should limit the number of threads building messages concurrently accordingly. Since the message buffers are an in memory storage area, the number of message buffers that can be created will be bound by available system memory.

Transferring messages

Transferring messages from a publisher to a subscriber is the second data processing stage. Minimizing or eliminating packet re-transmission due to dropped packets is the single most important factor in maximizing Sybase RAP performance.

Why packets are dropped

UDP packets can be dropped by the system for a number of reasons:

- The publisher is sending packets too fast for the network to keep up. In this case, the publishing socket `SO_SNDBUF` is too small, or the network is too slow, or the subscriber's socket `SO_RCVBUF` is too small, to keep up with the workload.

  The maximum values for `SO_SNDBUF` and SO_RCVBUF can be changed by reconfiguring the operating system. Sybase RAP publisher and subscriber software allocate the maximum buffer space allowed by the operating system for `SO_SNDBUF` and SO_`RCVBUF` buffering.

- The subscriber is not loading data as fast as the data is being received. This is likely due to CPU speed, or the number of CPUs on the subscriber system. This could also be due to an improperly tuned or configured ASE or Sybase IQ server, preventing data from loading as fast as possible.

Eliminating or reducing dropped packets

To reduce or eliminate dropped packets:

- Use multiple multicast channels. Each DataStream channel defined in a publisher or subscriber is a separate multicast channel. Each channel is a multicast IP address and port number pair, and contains its own operating system buffers and network interface.
- Use a high-speed network interface. The interface is specified when the channel is configured, and should reference the highest-speed network interface on your system. This interface should be at least 1gigabit/second (1000base-T/SX/LX).
- Use multiple network interfaces. If you have more than one network interface in your machine, you can configure one or more Sybase RAP DataStream Channels over each of them, for maximum transmission rates. This will allow greater throughput by the network, by splitting the data channels over two or more separate high-speed interfaces.
- Use multiple network interfaces. If you have more than one network interface in your machine, you can configure one or more Sybase RAP DataStream Channels over each of them, for maximum transmission rates. This will allow greater throughput by the network, by splitting the data channels over two or more separate high-speed interfaces.
- Use systems with more CPUs or cores, so that the multi-threaded subscriber can process incoming data in parallel. Each DataStream Channel is managed by its own thread, and separate threads are used to load each table in ASE or Sybase IQ. Therefore, there is ample opportunity to leverage parallel processing, if sufficient CPU and cores are available on the system.
- Configure your system so that the host computers for publishers and subscribers are on the same subnet, eliminating the need for packets to pass through routers.
- Configure your network to allow larger MTU sizes. The Maximum Transmission Unit (MTU) is the largest size of IP datagram which may be transferred in one frame using a specific data link connection, and can be configured to be larger than the default (which is typically 1500 bytes).

Increasing publisher performance

The following parameters defined in the configuration file *publisher.xml* affect the publisher performance.

Table 1-1: Publisher performance settings.

Parameter	Description
LogLevel	This parameter controls how much information is logged during system operation. Logging information is an expensive operation, and this level should be adjusted to log as little information as possible for effective operation of the system in your operational environment. For optimal performance this value should be set to error or warning.
NumPacketBuffers	This parameter controls the ability of the system to respond to a loss of information over the network. Specifically, this setting controls the number of packets to cache, on a per data stream basis, for use in responding to resend requests by a subscriber. Higher values of this setting will cause the system to allocate more memory to store previously sent information, and will increase the reliability of the system. This parameter should be tuned in conjunction with the MTU size of the host interface on which this publisher is operating. If the MTU size of the host interface is large, a much lower number of packet buffers will be required to achieve overall system reliability. It will also improve overall system performance.
DataStreamChannelList	The data stream channel is an independent pipeline through which messages are transmitted. Each data stream channel has its own set of packet, and network buffers. Adding multiple data streams to a publisher increases the amount of data that can be sent across the network reliably. This parameter should be increased in situations where the subscribers are not receiving the data sent by the publishers over the network, and when the subscribers themselves cannot keep up with the data sent by the publisher. Any changes made to the data channels in a publisher, must also be made to the subscribers who are subscribed to the publisher and may affect the performance of the subscribers.

Increasing subscriber performance

The following parameters defined in the configuration file *opensubscriber.xml* affect subscriber performance.

Table 1-2: Subscriber performance settings.

Parameter	Description
LogLevel	This parameter controls how much information is logged during system operation. Logging information is an expensive operation, and this level should be adjusted to log as little information as possible for effective operation of the system in your operational environment. For optimal performance this value should be set to error or warning.

Parameter	Description
NumPacketBuffers	Packet buffers are in memory buffers used on the subscriber side to store arriving packets until they are processed by the subscriber. The number of packet buffers is specified on a per data stream channel basis. This parameter should be increased if the message sending rate of the publisher exceeds the message receiving rate of the subscriber. If the subscriber can process data at a rate matching the data sent by the publisher, a value of 20-35 packet buffers per data stream channel is ample to allow for small bursts of increased traffic.
PacketWindowSize	The subscribers may receive network packets in an order other than the order the packets were sent. If a packet is received out of sequence, this parameter determines how many additional packets can be received before a resend request is sent for the missing packet. A very small value can result in requesting resends for packets that will arrive out of order due to network latencies. A very large value will increase the latency that occurs when packets are actually lost over the network. The packet window size is related to the number of packets cached by the publishers sending data to this subscriber (NumPacketBuffers). The packet window size of a subscriber should be 70-80% of the number of packets cached by the publisher to ensure that the publisher has the ability to send missing packets in response to resend requests. This setting should never be set to a value that exceeds the number of packet buffers allocated to its publisher as this will result in data loss in situations where resends are required.
DataStreamChannelList	This parameter lists the data stream channel definitions for the subscriber. The list should match the data stream channel list defined for the publishers to which this subscriber is subscribed. The number of data stream channels should be increased if the message sending rate of the publisher exceeds the message receiving rate of the subscriber. It should be noted, however, that a very large number of data channels has a negative impact on the overall system performance as it results in resource contention.

Loading messages

Loading messages into the target databases (RAPCache and RAPStore) is the final data processing stage. Subscribers convert messages into their native binary format before loading the messages.

RAPCache performance tuning

Tuning the RAPCache database can increase overall system performance. Increasing the number of partitions per database table, CPUs (engines) in the database server and the cache size can significantly improve the overall system performance. For detailed information about RAPCache configuration parameters and performance tuning suggestions, please refer to the ASE documentation.

Adjusting these configuration parameters in *rapcache.xml* can affect the RAPCache message loading performance.

Table 1-3: RAPCache performance options.

Parameter	Description
TDSPacketSize	The TDS packet size affects the maximum size of the packets that are transmitted between the RAPCachee subscriber and the RAPCache database. Larger TDS packet size (8K or 16K) values reduce network communication overhead between the subscriber and the cache and help improve system performance. To use a large TDS packet size, adjust these parameters in the RAPCache database: • Set the **additional network memory** parameter to allow for (TDSPacketSize x The number of loaders) additional memory. The number of loaders in the system will be the number of database tables x the number of partitions per table. • Set the **max network packet size** to allow for the packet size set in the RAPCache subscriber configuration. Additional information on these parameters can be found in the ASE documentation.
BulkInsertArraySize	This parameter affects the amount of buffer the RAPCache Subscriber will use before transferring data to the RAPCache. Large values for this parameter along with a large TDS packet size can result in reduced network overhead between the RAPCache and the subscriber. The value set for this parameter is constrained by overall system memory.
BulkBatchSize	Information loaded into the RAPCache is committed to the database periodically and made available to readers. This parameter controls how much information can be loaded into the RAPCache before it is committed. Higher values of this parameter increase the latency of the system, and require the number of locks in the RAPCache to be increased to accommodate larger database transactions. Smaller values for this parameter introduce the extra overhead of frequently committing transactions to the database and result in slower load performance.

Parameter	Description
IdleBufferWriteDelayMSec	This parameter determines the number of milliseconds to wait before data is transferred to the RAPCache when the system is not receiving data from the publisher. Low values of this parameter will ensure lower system latency in periods where the information flow is not steady.

RAPStore performance tuning

Adjusting these configuration parameters in *rapstore.xml* can affect the RAPStore message loading performance.

Table 1-4: RAPStore performance options.

Parameter	Description
PrimaryFileLocation & OverflowFileLocation	These parameters control where binary files containing data to be loaded into the RAPStore are stored. For optimal system performance, these files should be stored on separate physical disk controllers than the RAPStore system itself. As well, the primary and overflow locations should be placed on separate file systems.
IOBuffferSizeMB	This parameter determines the amount of data stored in memory per database table before the data is written out to a file. Larger values for this parameter result in higher system throughput, but the parameter is constrained by the amount of available system memory.
NumIOBuffers	IO buffers are in memory storage locations that allow the system to continue to receive data when it is writing previously received data out to a file. This parameter determines the number of IO buffers created per table. Increasing the number of IO buffers beyond a value of 3 or 4 does not increase the performance of the system.
TargetFileSizeMB	This parameter determines the size of each binary file that is loaded into the RAPStore. Larger values for this parameter will increase the latency of the system. Small values for this parameter will affect system performance due to the I/O consequences of writing out and loading many individual files.
IdleBufferWriteDelaySec	This parameter determines the number of seconds to wait before data is written out to file when the system is not receiving data from the publisher. Larger values for this parameter will caused increase latency in situations where data flow is not steady. Very small values will increase the file overhead costs in the system, and will reduce performance.

Message filtering

Both the RAPCache and RAPStore subscribers can be configured to automatically filter certain messages. A rule definition specifies a message type (by it's ID, which is set in the template for that message type), whether the rule is an include rule or an exclude rule, and the condition. For include rules, only messages that match the condition are loaded. For exclude rules, messages that match the condition are excluded.

It is not valid to define both an include rule and an exclude rule for the same message type. However, the same message type can have multiple rules of the same type. These rules are defined by using multiple Rule tag blocks in the configuration XML file, as shown in the sample file below.

Message filtering is configured in *messagefilter.xml*. This file lives in the *config* directory of the RAP Cache and RAP Store. The file should be identical for both subscribers. There is a messagefilter.xsd file that is in the same directory.

The file below is a sample. The file shipped with the product does not contain any rules. It contains only the *MessageFilter* and *RuleList* tags.

```
<?xml version="1.0" encoding="UTF-8"?>
   <MessageFilter>
      <RuleList>
         <Rule>
            <MessageType>4</MessageType>
            <RuleType>exclude</RuleType>
            <FieldRule>
               <FieldName>Exchange</FieldName>
               <FieldValue>NYSE</FieldValue>
            </FieldRule>
         </Rule>
         <Rule>
            <MessageType>13</MessageType>
            <RuleType>include</RuleType>
            <FieldRule>
               <FieldName>Symbol</FieldName>
               <FieldValue>SY</FieldValue>
            </FieldRule>
         </Rule>
      </RuleList>
</MessageFilter>
```

The following table explains the meaning of each of the XML elements in the template.

Table 1-5: Message Filterin Element Descriptions

Element	Description
MessageFilter	Root element for the message filters configuration file.
RuleList	A list of rules for filtering messages.
Rule	Contains conditions for a rule. Contains one *MessageType* element and zero or one *FieldRule* elements.
MessageType	The type of message to include/exclude.
RuleType	Indicates the type of rule. Valid values are: • `exclude` – if the rules holds true, exclude the message • `include` – if the rule holds true, include the message
FieldRule	Contains conditions for rules on a particular field.
FieldName	The name of the field to include/exclude when the value of the field matches *FieldValue*.
FieldValue	The value of the field to match. This value must be non-null. • If the value is a time, the format of the value must be `hh:mm:ss` or `hh:mm:ss.sss` • If the value is a date, the format must be `YYYY-MM-DD` • If the value is a *DateTime*, then the format must be `YYYY-MM-DDThh:mm:ss` or `YYYY-MM-DDThh:mm:ss.sss`

Use cases for feed handlers

This section provides several examples of use cases for feed handlers. Note that in the examples, error checking has been ignored for simplicity; production code should include error checking.

Sending a market data message

The Publisher API offers your methods that allow the feed handler to build a message to be sent to the Sybase RAP system. After initializing the Publisher API, do the following to build a message:

```
PUB_CALLBACKS * callbacks;
PUB_STARTUP * startup;
PUB_SEND_MESSAGE_CONTEXT * msg;
```

```
        uint16_t error_code;

            callbacks = Allocate a PUB_CALLBACKS structure;
            callbacks->shutdown = &myShutdownEventReceiver;

            startup = Allocate a PUB_STARTUP structure;
            startup->config_dir = configuration file directory or null;
            startup->template_dir = "templates";
            startup->strict_check = false;
            startup->component_subtype = "FAST Feed Handler";
            startup->callbacks = callbacks;

        pub_initialize( startup );
        msg = Allocate a PUB_SEND_MESSAGE_CONTEXT structure;
        loop for each message {
            error_code = pub_beginMessage( <message_type>, msg );
            error_code = pub_setInt32Field( msg, <field_name>, <field_value> );
            error_code = pub_setStringField( msg, <field_name>, <field_value> );
            error_code = pub_setInt16Field( msg, <field_name>, <field_value> );
            error_code = pub_sendMessage( msg );
}
            error_code = pub_flush();
            error_code = pub_shutdown( false );

        Free the PUB_SEND_MESSAGE_CONTEXT structure;
        Free the PUB_CALLBACKS structure;
```

To build multiple messages at once, allocate several `PUB_SEND_MESSAGE_CONTEXT` structures and use one structure per message being simultaneously built. Only one call should be made to initialize the Publisher, not one per thread. Similarly, only a single call should be made to shut down the Publisher.

The feed handler does not need to call flush. This call is optional. The shutdown API accepts a Boolean parameter which indicates whether to flush any buffered messages before shutting down.

Shutting down a feed handler

When the Publisher receives a shutdown request (likely initiated by the Operations Console), the Publisher calls the feed handler shutdown callback. The Publisher does not perform any shutdown action upon receipt of this request, since the feed handler may need to perform its shutdown actions first. The feed handler should perform its shutdown activities, call `cancelMessage` for any messages that are in progress of being built (or finish building them), and then call the Publisher shutdown method.

CHAPTER 2

RAP Data Stream Templates

About this Chapter

This chapter provides a description of the function and format of RAP Data Stream (RDS) templates to use for customizing template files for your particular data feed application.

Contents

Overview

The Publisher API and the subscribers look up information about message formats in a set of RAP Data Stream (RDS) templates. RDS templates are XML documents that represent the data structure of specific message types. The Publisher API uses templates to build the messages it sends to subscribers; subscribers use templates to parse messages and store them in a database.

The use of RDS templates minimizes the network bandwidth required for message transmission across the network. In addition, RDS templates increase processing efficiency by minimizing the number of CPU cycles needed to process each message.

Templates are located using a directory passed into the Publisher initialization method. All files that reside in that directory are read into memory. There may be one or more template files and each file may contain one or more message definitions. *templateprocessor.a* is the static library containing the Template Processing module that is shipped with Sybase RAP.

Notes

- See "RDS template" on page 53 for a sample RDS template file and list of XML element definitions.
- Publishers and subscribers can reside on different computers, but the same templates and same version of the software must be used by both the publisher and subscriber.
- All message types must be expressed in the proper template format. A template specifies all fields included in a message, as well as the sequence of those fields. Each field within a message must be set in the order in which it is defined in the template.

Datatype conversion

The RAPCache and RAPStore subscribers attempt automatic datatype conversion between the RAP Data Stream (RDS) datatypes used to transfer messages between the publisher and subscriber sides, and the database column datatypes used to store the information. In general, datatype conversion is attempted, if the types are compatible and no loss of precision occurs.

Supported datatypes

- 8, 16, 32 and 64-bit SIGNED INTEGER and UNSIGNED INTEGER
- NUMERIC and DECIMAL datatypes with precision from 1 to 18, inclusive
- DATE
- DATETIME
- CHARACTER strings less than or equal to 255 bytes

Note Due to the underlying differences between the RAPCache and RAPStore databases, if a message contains an empty string, the empty string is stored as NULL in the RAPCache database and is stored as an empty string in the RAPStore database.

Unsupported datatypes

Sybase RAP does not support RDS message definitions that reference a table which has a CHAR or VARCHAR column which is non-null and has a default value, where that column is not explicitly set by the RDS message. The following is a list of datatypes which are not supported by Sybase RAP:

- LOB (large object binary or text)
- MONEY datatype is not supported (however, SMALLMONEY is supported)
- FLOAT
- CHAR or VARCHAR datatypes with size greater than 255 have their data truncated on INSERT at 255 characters.
- BIT

Integer conversions

Any signed or unsigned integer can be converted to a larger signed integer. Only unsigned integers can be converted to a larger unsigned integer.

Any signed or unsigned integer can be converted into a NUMERIC or DECIMAL datatype with (precision minus scale) greater than or equal to precision required to represent the maximum value of the type of the integer. Unsigned 64-bit integers cannot be converted into numeric, as they require a numeric with precision at least 19, which is unsupported.

Numeric conversions

A NUMERIC or DECIMAL datatype with scale 0 can be converted into a SIGNED INTEGER, provided the precision required to represent the maximum value of the integer is greater than or equal to the precision of the NUMERIC or DECIMAL datatype.

NUMERIC or DECIMAL datatypes with a scale other than zero cannot be converted to an integer, and numerics cannot be converted into unsigned integers.

A NUMERIC or DECIMAL datatype can be converted to other numeric or decimal types, provided that the scale of the target type is greater than or equal to the scale of the source numeric, and the precision of the target numeric is greater than or equal to the precision of the source numeric plus the difference between the target scale and the source scale.

Examples

```
NUMERIC(4, 2) can be converted into a NUMERIC(4, 4)
NUMERIC(2, 2) can be converted into a NUMERIC(4, 4)
NUMERIC(2, 4) cannot be converted into a NUMERIC(4, 0)
NUMERIC(2, 4) cannot be converted into a NUMERIC(3, 6)
```

Date and time conversions

- TIME datatypes cannot be converted between types.
- DATE datatypes are converted to DATETIME by setting the time portion to midnight.
- DATETIME datatypes cannot be converted to dates or times.

Lookup tables

The lookup data is loaded once during initialization. The lookup column must be convertible into the RDS type of the lookup data. The lookup return column data type must be convertible into the data type of the destination column of the target table.

Recommended RDS datatype to SQL datatype mappings

For optimal performance, Sybase recommends that you map the following RDS datatypes to the following database column types.

Table 2-1: RDS datatype to SQL datatype mappings

RDS datatype	ASE datatype	Sybase IQ datatype
uint8	tinyint	tinyint
uint16	unsigned smallint	unsigned int (IQ has no unsigned smallint)
uint32	unsigned int	unsigned int
uint64	unsigned bigint	unsigned bigint
sint8	smallint (ASE has no signed 8-bit integer)	smallint (IQ has no signed 8-bit integer)
sint16	smallint	smallint
sint32	int	int
sint64	bigint	bigint
decimal(p, s)	numeric(p, s) or decimal(p, s)	numeric(p, s) or decimal(p, s)
datetime	datetime or smalldatetime	timestamp
date	date or smalldate	date
time	time	time
string	char(n) or varchar(n) n <255*	char(n) or varchar(n) n <255*

Notes

- If you know that data for a column is always be less than a specific length, then setting *n* to be as small as possible yields better performance.
- For maximum performance using the Publisher API to set decimal fields, the `pub_setDecimalFieldFromMantissa API` should be used instead of the `pub_setDecimal API`, if you can obtain the desired value in a format other than a double.

CHAPTER 3

Publisher API

About this Chapter

This chapter describes Publisher classes, objects, and methods.

Contents

Overview

The Publisher API is a mechanism that is invoked by market data feed handlers to build and process messages that allow users to publish data in a standard format.

Notes

- See "Publisher configuration" on page 48 for a sample *publisher.xml* template and list of XML element definitions.
- Subscribers must be running before the Publisher is started so that the subscribers are ready to receive messages when the Publisher sends them.
- If multiple Publishers are running, each instance must use a unique UDP broadcast address

Initialization

Initialization starts the publication mechanism by performing the following operations:

- Obtains configuration parameters from *publisher.xml* and establishes communications with subscribers.
- Provides the Sybase Unified Agent Framework (UAF) Interface. Operations Console uses the UAF to provide an agent for each host on which Sybase RAP components are installed. The UAF agent interfaces with the Operations Console, receiving commands and requests for component information and configuration. The UAF also forwards requests as needed to each Sybase RAP component installed on the host.
- Preallocates message buffers. The Publisher API accepts messages from the feed handler application and forwards them to subscribers. There are 2 types of buffering: the first is used to buffer messages, which saves memory allocation time for each new message; the second is for resend packets, which is used to resend packets.
- Initializes timing services. Initialization starts a timing service, which is a thread that sleeps or wakes according to an interval timer. This thread notifies a message send service when it is time to send a partially full buffer to ensure minimal latency of market data messages. The task also marks intervals for statistics collection.

- Initializes the heartbeat mechanism. In order to facilitate a highly-available configuration, each Sybase RAP component must respond to a heartbeat sent by a separate program. The sender of the heartbeat message uses the response (or lack thereof) to determine availability of the component. If necessary, a Sybase RAP FAILOVER is invoked. The heartbeat is initialized during `Init()` processing.
- Initializes statistics. The Publisher API manages statistics on message traffic and reports these statistics to UAF on request.
- Initializes the Resend infrastructure. If the Publisher API uses multicasting to publish messages, resending messages may be necessary, when asked to do so by any subscriber. The Publisher API caches candidates for resend in a circular buffer.

Configuration Property Values

The Publisher API enables access to configuration property values by name. Not all configuration properties may be intended for internal consumption by the Publisher API. Some may be intended for use by the feed handler application itself.

Message Description

The Publisher API code uses the field name to check that the SetField() functions have been called in the correct order and that the data value passed in is the correct data type.

Note Each field within a message must be set in the order in which it is defined in the template.

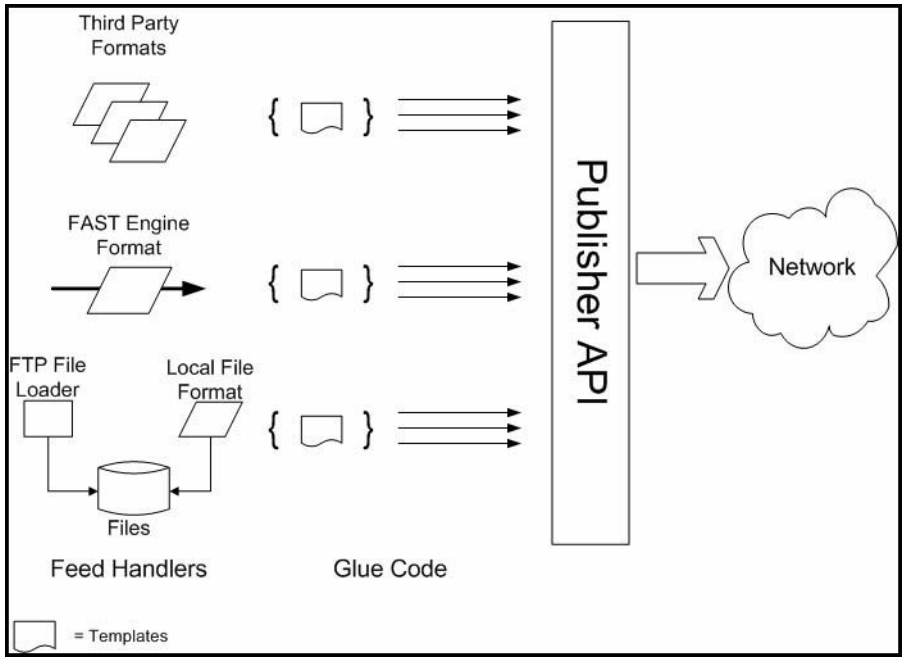
Begin Message

Begin message signals the Sybase RAP code that a new message is being started.

Each network buffer has one network buffer header to indicate the sequential packet number being sent. Buffering is necessary to ensure multicast ordering and to allow possible resend requests.

The Publisher API encodes each value sent to it via SetField() directly into the wire-ready format buffer. Each successive value of a message is directly appended to make one string to be sent over the network. Since messages may contain as few as 60 characters, multiple messages can be inserted into a single network packet buffer. MTU size is used to determine the packet size.

The following diagram illustrates the role of glue code using the feed handler SetField API to send data to the Publisher. The Publisher API then sends the code out on the network.



SendMessage

When the SendMessage() function is called, the buffer to be sent is already in the appropriate format. The message template number used to construct the message is the first field of the wire format message.

SendMessage() places one or more messages into one wire format network buffer.

When SendMessage() is called, it must have been configured to define the channels over which to multicast the packet. SendMessage() is also responsible for placing a sequential network packet number on each packet that it multicasts.

Publisher uses multiple channels for sending messages to maintain high performance through parallelism.

Returns SUCCEED or FAIL

Constants

Error Codes

The following constants are defined to represent error codes.

Constant	Description	Value
ERR_NONE	No error. Success.	0
ERR_PB_	Constants for error codes returned by the Publisher API are prefixed with ERR_PB.	2000-2999 This range is reserved for the use of the Publisher API.

Publisher API Data Structures

The following structure holds information for callbacks to the feed handler. Only one callback is listed.

```
typedef struct {
void (*shutdown)(void);
/* notifies caller of shutdown
                              request */
} PUB_CALLBACKS;
```

Initialization Information

The following structure holds information needed by the Publisher for initialization. See the initialization method for an explanation of the members of this class.

```
typedef struct {

/* directory in which to locate configuration file */
        const char * config_dir;
```

```
/* directory in which to locate RDS templates */
        const char * template_dir;

/* indicates what type of publisher this is */
        const char * component_subtype;
/* enables/disables strict checking of messages as
they're being built */
        bool  strict_check;
/* indicates whether the feed handler has initialized
the logger */
         bool own_logger;
/* callbacks to the feed handler */
        PUB_CALLBACKS callbacks;
} PUB_STARTUP;
```

Data Message Information

The following structure holds context information about a market data message that is being built to be sent to subscribers.

```
typedef struct {
/* MESSAGE_DEFN pointer */

void *    message_defn;
uint16_t  current_field_num;
/* location of destination message buffer */
        uchar * dest_buffer;
/* location of next fixed length value */
        uchar *  current_fixed_data;
/* location of next variable length value */
        uchar * current_variable_data
/* length of market data message */
        uint16_t message_length;}
PUB_SEND_MESSAGE_CONTEXT;
```

Methods

Description

The Publisher API is invoked by feed handlers to build and process messages. See the section entitled "Use cases for feed handlers" on page 12 for examples of how a feed handler can use the Publisher API.

The Publisher API uses the template processing module as its source of information about message formats.

Methods

- pub_initialize
- pub_beginMessage
- pub_cancelMessage
- pub_setField
- pub_sendMessage
- pub_flush
- pub_shutdown

pub_initialize

Description

Initializes the Publisher API. This method must be called before any other API in the Publisher API. This method should be called only once.

Syntax

```
uint16_t pub_initialize( PUB_STARTUP * startup_settings );
```

Parameters

PUB_STARTUP * startup_settings
Contains information needed by the publisher in order to initialize.

char * config_dir
The directory in which to locate the *publisher.xml* file.

char * template_dir
The directory in which to locate RDS templates.

bool strict_check
A value indicating whether strict checking should be performed on messages being built. True indicates that strict checking should be performed. This setting is recommended during development. False indicates that strict checking should not be performed. This setting is recommended in production.

bool own_logger
True if the feed handler has initialized the log file and false if the Publisher should initialize the log file.

char * *component_subtype*
A value indicating the type of publisher this is. For example, “Fast Feed Handler” or “Demo Feed Handler”. This information is used to identify the type of publisher when displaying information in the Operations Console.

PUB_CALLBACKS * callbacks

Callbacks used to notify the caller of the Publisher API of events. One callback is currently supported, which is a function to call if the Publisher receives a request to shutdown.

The caller of the Publisher API may need to perform shutdown activities that need to occur prior to the shutdown of the Publisher API. The Publisher API invokes this callback function to notify the caller that a request to shut down has been received. The caller is responsible for performing whatever actions need to be taken and then invoke the Publisher's shutdown API.

Return value

```
uint16_t error_code
```

An error code or ERR_NONE (value 0).

pub_beginMessage

Description

Indicates to the Publisher that a new message is being built. This method must be called before setting any of the fields on the message.

Syntax

uint16_t pub_beginMessage(uint16_t message_type, PUB_SEND_MESSAGE_CONTEXT * msg).

Parameters

uint16_t message_type

The type of message. This value needs to match the message type indicated in the template.

PUB_SEND_MESSAGE_CONTEXT * msg

The message context for the message being built. This structure must be allocated before calling this method. To build multiple messages at once from multiple threads, allocate several `PUB_SEND_MESSAGE_CONTEXT` structures and use one per message being simultaneously built.

Return value

```
uint16_t error_code
```

An error code or ERR_NONE (value 0).

pub_cancelMessage

Description

Indicates to the Publisher that a message that was being built is to be cancelled. This method is called after `beginMessage` to free up any resources being used by the message context.

Syntax

uint16_t pub_cancelMessage(PUB_SEND_MESSAGE_CONTEXT * msg);

Parameters

PUB_SEND_MESSAGE_CONTEXT * msg
The message context for the message being built.

Return value

```
uint16_t error_code
```

An error code or ERR_NONE (value 0).

pub_setField

Description

Sets the value of a field in a message that is being built.

Syntax

uint16_t pub_setInt8Field(PUB_SEND_MESSAGE_CONTEXT * msg, char * field_name, int8_t field_value);

uint16_t pub_setInt16Field(PUB_SEND_MESSAGE_CONTEXT * msg, char * field_name, int16_t field_value);

uint16_t pub_setInt32Field(PUB_SEND_MESSAGE_CONTEXT * msg, char * field_name, int32_t field_value);

uint16_t pub_setInt64Field(PUB_SEND_MESSAGE_CONTEXT * msg, char * field_name, int64_t field_value);

uint16_t pub_setUInt8Field(PUB_SEND_MESSAGE_CONTEXT * msg, char * field_name, uint8_t field_value);

uint16_t pub_setUInt16Field(PUB_SEND_MESSAGE_CONTEXT * msg, char * field_name, uint16_t field_value);

uint16_t pub_setUInt32Field(PUB_SEND_MESSAGE_CONTEXT * msg, char * field_name, uint32_t field_value);

uint16_t pub_setUInt64Field(PUB_SEND_MESSAGE_CONTEXT * msg, char * field_name, uint64_t field_value);

uint16_t pub_setDecimalField(PUB_SEND_MESSAGE_CONTEXT * msg, char * field_name, double field_value);

The following method accepts a mantissa (the significant digits) and an exponent. The value of the decimal is mantissa x 10^exponent.

uint16_t pub_setDecimalFieldFromMantissa(PUB_SEND_MESSAGE_CONTEXT * msg, char * field_name, int64_t mantissa, int8_t exponent);

uint16_t pub_setDateField(PUB_SEND_MESSAGE_CONTEXT * msg, char * field_name, uint32_t field_value);

uint16_t pub_setTimeField(PUB_SEND_MESSAGE_CONTEXT * msg, char * field_name, uint32_t field_value);

uint16_t pub_setDateTimeField(PUB_SEND_MESSAGE_CONTEXT * msg, char * field_name, uint64_t field_value);

uint16_t pub_setStringField(PUB_SEND_MESSAGE_CONTEXT * msg, char * field_name, uchar * field_value);

Parameters

PUB_SEND_MESSAGE_CONTEXT * msg
The message context for the message being built.

char * field_name
The name of the field being set. This name is only used when strict checking is on.

<data type> field_value
The value of the field which is to be placed into the message.

Date values
Express the number of days since year 0000.

Time values
Express the number of microseconds since midnight.

Datetime values
Consist of a 32-bit date value followed by a 32-bit time value.

String values
All string values are in UTF-8 format.

Return value

```
uint16_t error_code
```

An error code or ERR_NONE (value 0).

pub_sendMessage

Description

Sends a market data message. This method may or may not physically send a message. Messages are buffered until a packet is full, and then sent over the network.

Syntax

uint16_t pub_sendMessage(PUB_SEND_MESSAGE_CONTEXT * msg);

Parameters

PUB_SEND_MESSAGE_CONTEXT * msg
The message context for the message being built.

Return value

```
uint16_t error_code
```

An error code or ERR_NONE (value 0).

pub_flush

Description

Sends any buffered market data messages over the network to the subscribers.

Syntax

uint16_t pub_flush(void);

Return value

```
uint16_t error_code
```

An error code or ERR_NONE (value 0).

pub_shutdown

Description

Disconnects from the network and discards any resources being used by the Publisher API. This method is called when the feed handler is finished using the Publisher API.

Syntax

```
uint16_t pub_shutdown( bool flush );
```

Parameters

bool flush

Indicates whether any buffered messages should be sent before disconnecting.

Return value

```
uint16_t error_code
```

An error code or ERR_NONE (value 0).

CHAPTER 4

FAST Feed Handler

About this Chapter

This chapter describes Sybase RAP FAST Feed Message Handler classes, objects, and methods.

Contents

Overview

The FIX Adapted for Streaming (FAST) protocol is an emerging standard defined by the FIX Protocol, Ltd., as a method of compressing messages to be exchanged over a network connection.

The Sybase RAP FAST Feed Handler provides a mechanism for passing FAST-formatted data to the RAP Publisher. The Sybase RAP FAST feed handler connects directly to a market data feeds, receives FAST-encoded messages, decodes them, and makes the decoded data available to message handler plug-ins.

Notes

- The Sybase RAP FAST Feed Handler supports FAST SCP 1.1
- See "FAST Feed Handler configuration" on page 50 for a sample *fastfeedhandler.xml* template and list of XML element definitions.

Implementing a FAST feed handler

The Sybase RAP Message Handler receives decoded FAST messages and maps them into the RAP Publisher API. The library is compiled into a separate shared library which is dynamically loaded at runtime. The library exposes the symbols for the initialization, finalization, and process routines as C APIs so that they can be loaded from the FAST Feed Handler executable.

The implementation of the Sybase RAP Message Handler for FIX will be provided as a C++ class which is instantiated during the initialize function, a reference to which is stored in the `ffh_mh_info` structure so that it can be retrieved later during the process and finalize functions.

Sample code

```
/* Initialization function */

typedef int32_t (*ffh_mh_init_function)(

       ffh_mh_info    *,   /* handler info */
       ffh_init_param *,   /* array of init_param */
       int32_t );          /* num ffhinit_param items */

/* Finalization function */
```

```
typedef void (*ffh_mh_fini_fuction)( ffh_mh_info * );

/* Process Function */

typedef int32_t (*ffh_mh_process_function)( ffh_mh_event * );

/* Return value from the above functions which retun int32_t indicating success.
Any other values indicate an error and are defined internally by the library. */
#define FFH_MH_SUCESS      0

/* Callback functions */
typedef void (*ffh_mh_p_session_shutdown)( ffh_mh_info * );
typedef void (*ffh_mh_p_release_message)( ffh_mh_info *,
       ffh_fast_message * );
typedef struct {const char * name;const char * value;}
       ffh_init_param;

typedef enum ffh_mh_event_type {MHET_RECEIVE_MESSAGE = 0};

typedef struct {
       int16_t version;
       void * handlerID;
       void * loggerID;
       ffh_mh_p_session_shutdown  shutdown;
       ffh_mh_p_release_message   release_message;
       void * user_data;

} ffh_mh_info;

typedef struct {

       int16_t             version;
       ffh_mh_event_type   event;
       ffh_mh_info       * info;
       ffh_fast_message  * message;
}      ffh_mh_event;
```

FAST feed handler API interface

Description

The following C API defines the function signatures that you must implement to build a message handler.

Methods

- initializeRAPHandler
- finalizeRAPCallback
- processRAPCallback

- initialize
- publisherShutdown

initializeRAPHandler

Description

Create and initialize the Sybase RAP message handler. This method is invoked from the FAST Feed Handler code by dynamically looking up this function pointer in the message handler shared library.

Syntax

```
extern “C” int32_t initializeRAPHandler
( ffh_mh_info    * info,
ffh_init_param * initParams,
int32_t          initParamLen );
```

Parameters

`ffh_mh_info * info`
the info structure for calls to the message handler

`ffh_init_param * initParams`
the initialization parameters for the handler

`int32_t initParamLen`
the number of elements in the initParams array

Return value

int32_t 0 if the initialization succeeded, non *0* error code otherwise.

Usage

1 Create a new instance of the RAP message handler, and assign it to the user_data field of the `ffh_mh_info`.

2 Invoke the initialize method on the RAP message handler and return the result.

finalizeRAPCallback

Description

Invoke the Message Handler finalize method and clean up any remaining resources. This method is invoked from the FAST Feed Handler code by dynamically looking up this function pointer in the message handler shared library.

Syntax

```
extern “C” void finalizeRAPHandler
( ffh_mh_info * info );
```

Parameters

ffh_mh_info * info
the info structure for calls to the message handler

Usage

1 Cast the user_data field of the `ffh_mh_info` to a RAP message handler object.

2 Invoke the finalize method on the message handler object.

3 Delete the handler object and set the user_data pointer in the `ffh_mh_info` to null.

processRAPCallback

Description

Invoke the Message Handler process method. This method is invoked from the FAST Feed Handler code by dynamically looking up this function pointer in the message handler shared library.

1 Cast the user_data field of the `ffh_mh_info` to a RAP message handler object.

2 If the event type is `MHET_RECEIVE_MESSAGE`, invoke the receiveMessage method on the message handler object and return the result.

Syntax

extern “C” int32_t processRAPCallback (ffh_mh_event * event);

Parameters

`ffh_mh_event * event`
the event to be processed

Return value

`int32_t error_code`

initialize

Description

Initialize the Message Handler with any necessary setup actions. In particular, initialize the publisher API. The following table gives the initialization parameters required to be in the message handler definition in the FAST Feed Handler main configuration file.

Name Value	Description
`PublisherConfigFile`	This should point at the configuration file for the publisher.

Syntax

```
int32_t initialize(
        ffh_mh_info * info
        ffh_init_param * initParams,
        int32_t initParamLen
);
```

Parameters

PublisherConfigFile
This should point at the configuration file for the publisher.

ffh_mh_info * info
the info structure for calls to the message handler

ffh_init_param * initParams
the initialization parameters for the handler

int32_t initParamLen
he number of elements in the initParams array

Return value `int32_t 0` if initialization succeeded, `error code != 0` otherwise.

Usage

1 Initialize the RAP Publisher API, passing a reference to the static publisherShutdown method and pass in the `loggerID` pointer. If an error occurs, log it and return false.

2 Create a new RAP publisher send message context.

3 Store the `ffh_mh_info` in the provided pointer.

4 Return `0`.

receiveMessage

Description Process a decoded FAST message by mapping it into the publisher API.

Syntax

Parameters `ffh_mh_info * info`
the info structure for this message handler

ffh_fast_meesage * fastMsg
the decoded FAST message

Return value `int32_t 0` if processing succeeded; `error code != 0` otherwise

Usage This method will contain hard-coded entries for processing specific FIX messages. The general semantics are as follows:

1 Walk the FIX message.

- If it is a message that is recognized, store any data required for a RAP Publisher message, and once a RAP Publisher message is complete, invoke the publisher begin message API, plug in the available data, and call the publisher send message API.
- Repeat for each Publisher message contained within the FIX message. Details on the mappings can be found in Section 9 below. In the event that the FIX message is not supported, log a message indicating that the message type was not processed. In the event of an error from the publisher API, log the error and call the publisher cancelMessage API.

2 Once the entire FIX message has been processed (or if the message was not processed, or if an error has occurred), invoke the releaseMessage callback from the handler info structure.

finalize

Description: Clean up any resources allocated in the initialize method.

Syntax: void finalize();

Usage:

1 Call the publisher API shutdown method.

2 Free the send message context.

3 Clean up any remaining data structures.

publisherShutdown

Description: This is the callback method required by the Publisher API in order to indicate that a shutdown has been requested.

Syntax: static void publisherShutdown();

Usage: If the `ffh_mh_info structure` is not null, invoke the structure session shutdown function, passing in the `ffh_mh_info`.

CHAPTER 5

Logging

About this Chapter

This chapter briefly describes the Logging API interfaces.

Contents

Overview

Sybase RAP uses *log4cxx* version 1.2, an open source logging API, to log events. Log4cxx allows you to control which log statements are output. You can find information about log4cxx at http://logging.apache.org/log4cxx/index.html.

The logging library is made available to feed handler developers so that you can use the same logging utility as the Sybase RAP modules. This allows events from custom feed handlers to be logged to the same log file as the file being used by Sybase RAP modules.

Note At higher log levels, more log information is reported. Processing is slower to execute at higher levels.

Logging levels

Log4cxx has 7 log levels. Sybase RAP maps these to 4 log levels, defined below, that are internally mapped to standard log4cxx. You should use these Sybase RAP log levels during feed handler development.

An include file, *logger.h* defines the following log levels, from highest to lowest:

```
LOGGER_DEBUG
LOGGER_INFO
LOGGER_WARNING
LOGGER_ERROR
```

Logging API interfaces

Description

The following API functions are declared in *logger.h*. Modules requiring logging must include this header file. The header file contains the datatype:

Methods

- log_open
- log_message
- log_close

- log_init_from_context
- log_get_context
- log_hexdump
- log_message_force

In addition to the method references, *logger.h* also contains the datatype:

```
typedef struct logger_context logger_context;
```

log_open

Description

Initializes RAPLogging. This method must be called before any other API in the RAPLogging API. This method should be called only once.

Syntax

```
uint16_t log_open( char * filepath , uint16_t log_level);
```

Parameters

char * filepath

The name and location of the log file. The file path can be either relative or absolute.

Uint16_t log_level

The log level defined in the configuration file. The log_level acts as filter, any message with lower log level will be ignored and not written into the log file. It can be defined in the configuration file as default filter value, or you can explicitly set the filter level dynamically.

Return value

uint16_t error_code (an error code) or *ERR_NONE (value 0)*.

log_message

Description

Writes the logging message and level into the log file.

Syntax

```
uint16_t log_message( uint16_t log level, uint16_t err_number, char * message);
```

Parameters

log level

Only the log levels defined in the "Logging levels" section are valid values.

err_number

Error number.

char * message

Message to be written out.

Return value — *uint16_t error code* (an error code) or *ERR_NONE (value 0)*.

log_close

Description — Close the log file and log hierarchy. The clog_close method must be called after other log operations and must be called only once.

Syntax — vuint16_t log_close();

Return value — *uint16_t error_code* (an error code) or *ERR_NONE (value 0)*.

log_get_context

Description — Returns a pointer to the logger_context used by current `RAPLogging` instance. This pointer can be used to initialize the `RAPLogging` in a shared library. For a description of steps needed to initialize `RAPLogging` in a shared library, see the "log_init_from_context"section.

Syntax — logger_context * log_get_context( );

Return value — *logger_context ** A pointer to the `logger_context` used by the current `RAPLogging` instance.

log_init_from_context

Description — Initializes the `RAPLogging` from an existing context that was created by a previous call to `log_open()`. This can be used to pass a logger instance to a shared library function. This function must be called before any other API in the `RAPLogging API` is invoked in the shared library.

The following sequence of calls should be used in your main executable to pass a logger instance to a shared library:

- Initialize the logger as usual by invoking log_open()
- Get the pointer to the logger_context by invoking log_get_context()
- Invoke your shared library function passing it the logger_context pointer.

In your shared library function:

- Invoke log_init_from_context() passing it the logger_context pointer as the argument.

- Invoke the log_message() function to log messages from your shared library.

Syntax

```
uint16_t log_init_from_context( logger_context * ctx  );
```

Parameters

logger_context * ctx
The logger_context pointer to be used to initialize the RAPLogging in a shared library.

Return value

uint16_t error_code (an error code) or *ERR_NONE (value 0)*.

log_message_force

Description

Write the logging message and level into the log file. This method forces the message into the log file regardless of the filter value of the log level. The filter value can be set in the configuration file (RAP) or explicitly set by calling log_open() on the feed handler side.

Syntax

```
uint16_t log_message_force( uint16_t level, uint16_t err_num, const char * message );
```

Parameters

log level: error number.
Only the values defined in section "Logging levels" are valid. For this method, log level is used primarily as informational display in the log file, as the message will be always be written into the log file.

err_number
error number.

char * message
The message to be written out to the log.

Return value

uint16_t error code (an error code) or *ERR_NONE (value 0)*.

log_hexdump

Description

Format and log a traditional memory dump starting at the address supplied and for the length supplied. Use the supplied context string at the top and bottom to make it easy to identify. Uses the internal logger-> debug function to write it to the current log file.

A typical dump file (with the date, time and thread values removed) looks like the following:

```
DEBUG rap4 - =====      DSHTable in Loader:Run New Work    ========
DEBUG rap4 - 26583E0 >B0236800 00000000 C0C16502 00000000< .#h.......e..... 00000000
DEBUG rap4 - 26583F0 >36000700 00000000 90846502 00000000< 6.........e..... 00000010
DEBUG rap4 - 2658400 >58886702 00000000 F8C86E00 00000000< X.g.......n..... 00000020
DEBUG rap4 - 2658410 >F07E6502 00000000 707F6502 00000000< .~e.....p.e..... 00000030
DEBUG rap4 - 2658420 >687E6502 00000000<                   h~e.....         00000040
DEBUG rap4 - =====      DSHTable in Loader:Run New Work    ========
```

Syntax

log_hexdump (char * contextString, void * address, long length)

Parameters

contextString
: The string printed at the beginning and end of the trace segment for identification.

address
: The beginning of the memory to dump.

length
: The number of bytes to log.

Return value

uint16_t error_code (an error code) or *ERR_NONE (value 0*).

APPENDIX A

Sample Configuration and Template Files

Refer to this appendix to review the structure of the configuration and template files.

Publisher configuration

The file *publisher.xml* is an XML document that contains configuration settings for the Publisher. A sample template is shown below. Each of the elements that is allowed in the template is described in the table that follows the sample. An XML schema (*.xsd*) describing the XML format is shipped as part of the product.

```
<?xml version="1.0" encoding="UTF-8"?>
<Publisher>

<Logger>
    <LogLevel>error</LogLevel>
     <LogFile>Publisher.log</LogFile>
</Logger>

<NumMessageBuffers>10</NumMessageBuffers>
<NumPacketBuffers>250</NumPacketBuffers>
<MessageFlushInterval>1</MessageFlushInterval>
<LatencyCheckInterval>30</LatencyCheckInterval>

<AdminChannel>
    <AdminPort>10001</AdminPort>
</AdminChannel>

<ResendChannel>
    <ResendPort>10002</ResendPort>
</ResendChannel>

<DataStreamChannelList>
    <DataStreamChannel>
       <ChannelName>Channel 1</ChannelName>
       <LocalInterface>localhost</LocalInterface>
       <IPAddress>224.0.0.1</IPAddress>
       <Port>12001</Port>
    </DataStreamChannel>
    <DataStreamChannel>
       <ChannelName>Channel 2</ChannelName>
       <LocalInterface>localhost</LocalInterface>
       <IPAddress>224.0.0.2</IPAddress>
       <Port>12002</Port>
    </DataStreamChannel>
</DataStreamChannelList>
</Publisher>
```

Publisher element descriptions

See the table below for Publisher template element descriptions.

Table A-1: Publisher XML element definitions

Element	Description
Publisher	Root element for the configuration file.
Logger	Contains settings for logging activities
LogLevel	The level of logging. Valid values are: error: only log errors warning: log warnings in addition to errors info: log informational messages in addition to messages logged at the warning level debug: log debugging messages in addition to messages logged at the info level
LogFile	The name and location of the log file. The file name can be relative or a full path.
NumMessageBuffers	The number of message buffers. One message buffer is required for each message that is being simultaneously built. This setting can have a value from 1 to 65535, though the machine must have enough memory to hold the number of buffers specified.
NumPacketBuffers	The maximum number of packets to cache in order to satisfy requests by a subscriber to resend a packet. This number of packets is cached per data stream channel. This setting can have a value from 1 to 4 billion, though the machine must have enough memory to hold the number of packets specified. This number of buffers is allocated on initialization of the Publisher.
MessageFlushInterval	The number of seconds after which to send any buffered messages. This setting can have a value from 1 to 65535.
LatencyCheckInterval	The number of seconds after which to do a latency check on a message. This setting can have a value from 1 to 65535.
AdminChannel	Information about the administration channel. This channel accepts requests for version information, statistics, and shutdown.
AdminPort	The port on which the Publisher will listen for incoming administration requests.
ResendChannel	Information about the resend channel. This channel listens for connections from subscribers. Subscribers will open a connection to a publisher and issue requests to resend packets.
ResendPort	The port on which the Publisher will listen for incoming connection requests from subscribers.
DataStreamChannelList	A list of data stream channel definitions. There can be up to 255 data stream channels.
DataStreamChannel	Contains information for one data stream channel. This name is used to identify the channel when logging.
ChannelName	A descriptive name for the channel.
LocalInterface	The local interface over which UDP packets will be sent.

Element	Description
IPAddress	The IP address under which messages will be broadcast.
Port	The port over which messages will be broadcast.

FAST Feed Handler configuration

The *fastfeedhandler.xml* configuration file stores the information needed to initialize the Sybase RAP FAST Feed Handler. See the XML file and table below for a description of this file. A sample template is also shown below. Each of the elements that is allowed in the template is described in the table that follows the sample. An XML schema (*.xsd*) describing the XML format ships as part of the product.

```
<?xml version="1.0" encoding="UTF-8"?>
<FASTFeedHandler>
    <Logger>
       <LogLevel>debug</LogLevel>
       <LogFile>FASTFeedHandler.log</LogFile>
    </Logger>

<FASTTemplateLibrary>
   <TemplateLibraryFile>
     filename.xml
   </TemplateLibraryFile>
    ...
</FASTTemplateLibrary>
<SessionManager>
    <FASTDataMode>stream</FASTDataMode>
</SessionManager>

<MessageHandler>
   <SharedLibrary>
      ffhfixmsghandler.so
   </SharedLibrary>
   <InitFunctionName>
      initializeRAPHandler
   </InitFunctionName>
   <FiniFunctionName>
       finalizeRAPHandler
   </FiniFunctionName>
   <ProcessFunctionName>
       processRAPCallback
```

```
    </ProcessFunctionName>

    <InitParams>
      <InitParamname="paramName">
          paramValue
      </InitParam>
        ...
    </InitParams>
</MessageHandler>

<Session>
    <SessionName>MySession</SessionName>
    <SessionConnection>
        <ListenHost>localhost</ListenHost>
        <MulticastHost>299.30.215.22</MulticastHost>
        <ListenPort>8899</ListenPort>
     </SessionConnection>
</Session>

</FASTFeedHandler>
```

FAST Feed Handler element descriptions

See the table below for FAST Feed Handler template element descriptions.

Table A-2: FAST Feed Handler element descriptions

Element	Description
FASTFeedHandler	This is the root element of the configuration file.
Logger	This groups log related information.
LogLevel	The level of logging. Valid values are: error: only log errors warning: log warnings in addition to errors info: log informational messages in addition to messages logged at the warning level debug: log debugging messages in addition to messages logged at the info level
LogFile	The file where the log should be written.
FASTTemplateLibrary	This defines a FAST template library. The library must contain at least one TemplateLibraryFile tag. If duplicate FAST templates are present within the definition of the library, the definition which appears last will be used.
TemplateLibraryFile	This tag specifies an absolute or relative location of a file which is in the FAST template XML format, or the absolute or relative location of a directory containing files which are in the FAST template XML format. The tag may be repeated 1 or more times in a template library.

Element	Description
MessageHandler	This tag contains configuration parameters related to the message handler.
SharedLibrary	This is the name of the shared library to load for the message handler.
InitFunctionName	This is the name of the initialization function to use for the message handler from the handlers shared library.
FiniFunctionName	This is the name of the finalization function to use for the message handler from the handlers shared library.
ProcessFunctionName	This is the name of the process function to use for the message handler from the message handlers shared library.
InitParams	This is a list of initialization parameters to be passed to the initialization function.
InitParam	This is an individual initialization parameter which provides the name and value which will be passed as strings to the initialization function.
SessionManager	This defines a session manager.
FASTDataMode	This determines whether the FAST data being received is using stream mode or block mode. Valid values are stream and block.
Session	This provides information on how the UDP connection should be established.
SessionName	This is the name of the session.
SessionConnection	A group of session connection information.
ListenHost	This is the local interface that a listener port should be opened on.
MulticastHost	This is the multicast interface that a listener port should be opened on. It is optional, and if omitted, socket will not accept multicast packets.
ListenPort	This is the local port that should be used for incoming messages.

RDS template

The standard RDS template file, *template.xml*, defines the information required by the Publisher API and subscribers. Since the subscriber allows lookups of a field value from a database table before inserting the value into the Sybase RAP schema, the template also allows the information required for this lookup to be defined.

A sample template is shown below. Each of the elements that are allowed in the template is described in the table that follows the sample. An XML schema (*.xsd*) describing the XML format is shipped as part of the product.

```
<Template>
    <MessageDefnList>
        <MessageDefn>
            <MessageDesc>Stock Quote</MessageDesc>
            <MessageType>1</MessageType>
            <DestTableName>STOCK_QUOTE</DestTableName>
            <FieldDefnList>
                <FieldDefn>
                    <FieldName>Instrument ID</FieldName>
                    <IntegerField>
                        <IntegerDataType>
                            uint32
                        </IntegerDataType>
                    </IntegerField>
                    <DestColumnName>
                        INSTRUMENT_ID
                    </DestColumnName>
                    <Lookup>
                        <LookupTableName>
                            INSTRUMENT
                        </LookupTableName>
                        <LookupColumnName>
                            INSTRUMENT_ID
                        </LookupColumnName>
                        <LookupColumnReturn>
                            INSTRUMENT_NAME
                        </LookupColumnReturn>
                    </Lookup>
                </FieldDefn >
                <FieldDefn>
                    <FieldName>
                        Quote Date
                    </FieldName>
                    <DateField/>
```

```
                        <DestColumnName>
                            QUOTE_DATE
                        </DestColumnName>
                    </FieldDefn>
                    <FieldDefn>
                        <FieldName>Quote Time</FieldName>
                        <TimeField/>
                        <DestColumnName>
                            QUOTE_TIME
                        </DestColumnName>
                    </FieldDefn>
                    <FieldDefn>
                        <FieldName>
                            Trading Symbol
                        </FieldName>
                        <StringField/>
                        <DestColumnName>
                            TRADING_SYMBOL
                        </DestColumnName>
                    </FieldDefn>
                    <FieldDefn>
                        <FieldName>Ask Price</FieldName>
                        <DecimalField>
                            <Precision>10</Precision>
                            <Scale>2</Scale>
                        </DecimalField>
                        <DestColumnName>
                            ASK_PRICE
                        </DestColumnName>
                    </FieldDefn>
                </FieldDefnList>
            </MessageDefn>
            <MessageDefn>
                ...
            </MessageDefn>
        </MessageDefnList>
    </Template>
```

RDS template XML element descriptions

See the table below for RDS template XML element descriptions.

Table A-3: RDS Template XML element descriptions.

Element	Description
Template	Root element for the template.
MessageDefnList	Contains a list of one or more message definitions.
MessageDefn	Contains information that defines a single message type.
MessageDesc	A description of the type of market data message. For example, Stock Quote. This element is used purely for descriptive purposes. The value can contain any string.
MessageType	A unique number representing the type of market data message. This number must uniquely identify the message type across all message definitions within all templates. The value can contain any integer from 1 to 65535.
DestTableName	The name of the database table into which the message should be stored. There is one database table per message type. The value can contain any string.
FieldDefnList	Contains a list of one or more field definitions.
FieldDefn	Contains information that defines a single field.
FieldName	Name of the field. The value can contain any string.
IntegerField	Indicates that the field is some type of integer. The field definition will only contain one of the following: IntegerField, DecimalField, StringField, DateField, TimeField, DateTimeField.
IntegerDataType	The data type of an integer field. Valid values are: uint8, uint16, uint32, uint64, sint8, sint16, sint32, sint64
DecimalField	Indicates that the field is a decimal value. The field definition will only contain one of the following: IntegerField, DecimalField, StringField, DateField, TimeField, DateTimeField.
Precision	The precision of a decimal field.
Scale	The scale of a decimal field (the number of digits after the decimal point).
StringField	Indicates that the field is a string. The field definition will only contain one of the following: IntegerField, DecimalField, StringField, DateField, TimeField, DateTimeField.
DateField	Indicates that the field is a date. The field definition will only contain one of the following: IntegerField, DecimalField, StringField, DateField, TimeField, DateTimeField.
TimeField	Indicates that the field is a time. The field definition will only contain one of the following: IntegerField, DecimalField, StringField, DateField, TimeField, DateTimeField.

Element	Description
DateTimeField	Indicates that the field is a datetime. The field definition will only contain one of the following: IntegerField, DecimalField, StringField, DateField, TimeField, DateTimeField.
DestColumnName	The name of the column into which the field data should be stored. There is one column per field. The value of this element can contain any string.
Lookup	Indicates that the data in the field should be used as a lookup for another table. This is an optional element. If it does not appear in a field definition, then no lookup is required.
LookupTableName	The name of the table to use to look up a value.
LookupColumnName	The name of the column to use to look up a value.
LookupColumnReturn	The name of the column from which to return data when doing a lookup.

CHAPTER 6

FIX Message to data model mappings

About this Chapter

Contents

This chapter identifies FIX Message to data model mappings.

General processing notes

The message mappings defined here use the FIX field names. For details on the actual field numbers and allowed values, see the FIX protocol specification version 4.4 at http://fixprotocol.org/specifications/fix4.4fiximate/index.html.

Instrument blocks

Each of the messages described below contains at least one instrument block that provides information on the entity for which data is present in the message. Some messages can contain information on multiple instruments. In general, the Symbol or Security ID fields of the instrument block map onto the stock symbol of the corresponding Sybase RAP table, and this is used to automatically look up instrument IDs in the instrument table. The lookup is completed via the RDS template mechanism as described in Chapter 2, "RAP Data Stream Templates."

Note A default value for lookup table failures must be specified by the site in the template.

Data aggregation and missing data

The data in FIX messages can be unordered and many FIX fields in a message are optional. The RDS format does not support optional values. Thus, it may be necessary to perform the following steps before invoking the Publisher API to send data:

1 Iterate over the available message and fill in a data structure representing the required data.

2 If, after processing, the FIX message does not contain sufficient data, log an error.

3 If the message contains enough data, then call the Publisher API to begin processing the correct RDS message, set the field data, and then send the message.

These steps should be performed on a per-instrument basis for FIX messages that support multiple instruments. For FIX messages that support multiple destination RDS messages for a single instrument, it may be necessary to aggregate data for multiple RDS messages concurrently, since the FIX data for the different RDS messages may be interleaved.

RDS message types

Since RDS does not support optional fields and not all of the FIX data mappings defined in this chapter contain all of the data to fully populate the standard RDS messages, it is necessary to define RDS message types which are similar to the full RDS messages, but have fields omitted.

Determining instrument type

Some FIX messages apply to different instruments, and determining which RDS messages to generate depends on the type of the instrument being processed.

Stock Instruments

If the Product field of the Instrument block is present and has the value 5 (EQUITY), then the instrument is a stock.

If Product is not present but Security Type is present and has the value CS or PS (common stock or preferred stock), then the instrument is a stock.

If both Product and Security Type are not present and the CFICode field is present and the first character of the field is E (for equity) and the second letter is one of S, P, R, C, F, or V, then this is a stock.

Otherwise, this is not a stock.

Note Sybase RAP - The Trading Edition does not support multi-leg securities.

For more information, see CFI Code values at http://www.iso.org/iso/catalogue_detail?csnumber=32835 for full documentation.

Bond instruments

If the CFICode is present and the first character of the field is D (for debt) and the second character is B (for bond), then the instrument is a bond.

Mutual fund instruments

If the Security Type is present and has the value of MF, then this is a mutual fund instrument.

Otherwise, if the Security Type is not present and the CFICode is present and the first character of the field is E (for equity) and the second is U (units), then this is a mutual fund.

Option instruments

If the Security Type is present and has the value of OPT, then the instrument is an option instrument.

Otherwise, if the Security Type is not present and the CFICode is present and the first character of the field is O (for options), then the instrument is an option.

Index instruments

If the Product field of the Instrument block is present and has the value 7 (INDEX), then the instrument is an index.

If Product is not present, but Security Type is present and has the value INDEX, then the instrument is an index.

If both Product and Security Type are not present and the CFICode field is present and the first character of the field is M (for miscellaneous), and the second letter is one of R (for referential instruments) and the third letter is I (for indices), then the instrument is an index.

Otherwise, this is not an index.

Advertisement FIX message

Stock trade RDS mapping

Table	STOCK_TRADE
Missing Data	None
Condition	The TradeDate field must exist in FIX message and this must be a stock instrument. See “Stock Instruments” on page 61 for more information.

FIX Message Field	Table Column
SecurityID/Symbol	INSTRUMENT_ID
TradeDate	TRADE_DATE
AdvID	TRADE_SEQ_NBR
SecurityID/Symbol	TRADING_SYMBOL
TransactTime	TRADE_TIME
Price	TRADE_PRICE
Quantity	TRADE_SIZE

Mutual fund history RDS mapping

Table	MUTL_FUND_HIST
Missing Data	None
Condition	The TradeDate field must exist in FIX message and this must be a mutual fund instrument. See see “Mutual fund instruments” on page 62 for more information.

FIX Message Field	Table Column
SecurityID/Symbol	INSTRUMENT_ID
TradeDate or current date	TRADE_DATE
SecurityID/Symbol	TRADING_SYMBOL
Price	PRICE

Bond trade RDS mapping

Table	BOND_TRADE
Missing Data	None
Condition	The TradeDate field must exist in FIX message and this must be a bond instrument. See “Bond instruments” on page 62 for more information.

FIX Message Field	Table Column
SecurityID/Symbol	INSTRUMENT_ID
TradeDate	TRADE_DATE
AdvID	TRADE_SEQ_NBR
Yield	TRADE_YIELD
TransactTime	TRADE_TIME
Price	TRADE_PRICE
Quantity	TRADE_SIZE

Option trade RDS mapping

Table	OPTION_TRADE
Missing Data	None
Condition	The TradeDate field must exist in FIX message and this must be an option instrument. See “Option instruments” on page 62 for more information.

FIX Message Field	Table Column
SecurityID/Symbol	INSTRUMENT_ID
TradeDate	TRADE_DATE
AdvID	TRADE_SEQ_NBR
There is no OPEN_INTEREST in Advertisement	OPEN_INTEREST
TransactTime	TRADE_TIME
Price	TRADE_PRICE
Quantity	TRADE_SIZE

Mass quote FIX message

Mass Quote FIX messages contain repeating groups for instruments. For each repeated grouping that instrument can have a repeated block of quote data. Each one of these quotes is processed in the same manner that a Quote FIX message is processed, as described below.

Stock quote RDS mapping

Table	STOCK_QUOTE
Missing Data	None
Condition	This must be a stock instrument. See "Stock Instruments" on page 61 for more information.

FIX Message Field	Table Column
SecurityID/Symbol	INSTRUMENT_ID
TransactTime	QUOTE_DATE
QuoteID	QUOTE_SEQ_NBR
SecurityID/Symbol	TRADING_SYMBOL
TransactTime	QUOTE_TIME
OfferPx	ASK_PRICE
OfferSize	ASK_SIZE
BidPx	BID_PRICE
BidSize	BID_SIZE

Bond quote RDS mapping

Table	BOND_QUOTE
Missing Data	None
Condition	This must be a bond instrument. See "Bond instruments" on page 62 for more information.

FIX Message Field	Table Column
SecurityID/Symbol	INSTRUMENT_ID
TransactTime	QUOTE_DATE
QuoteID	QUOTE_SEQ_NBR

FIX Message Field	Table Column
Yield	YIELD
TransactTime	QUOTE_TIME
OfferPx	ASK_PRICE
OfferSize	ASK_SIZE
BidPx	BID_PRICE
BidSize	BID_SIZE

Security status fix message

Stock history RDS mapping

Table	STOCK_HISTORY
Missing Data	OPEN_PRICE, CLOSE_PRICE
Condition	This must be a stock instrument. See “Stock Instruments” on page 61 for more information.

FIX Message Field	Table Column
SecurityID/Symbol	INSTRUMENT_ID
TransactTime	TRADE_DATE
SecurityID/Symbol	TRADING_SYMBOL
LowPx	LOW_PRICE
HighPx	HIGH_PRICE
BuyVolume/SellVolume	VOLUME

Installing a signal handler

The following describes how to install a signal handler to handle SIGINT (signal 3, or Ctrl-C) interrupts.

The following installs the signal handler:

```
struct sigaction sigAction;
```

```
sigAction.sa_handler = sigIntHandler;
sigaction( SIGINT,&sigAction, NULL );
```

The following is the actual handler code itself. For the FAST Feed Handler, the handler simply sets the global active flag to false. Other programs which need to use similar code may choose to signal or broadcast to a mutex.

```
static void sigIntHandler( int signum ) {
    active = false;
}
```

Index

R

S

U

X