FIX Shared Library

email:
support@aquaq.co.uk

web:
www.aquaq.co.uk

AquaQ Analytics

Revision History

Revision	Date	Author(s)	Description
0.1	May 6, 2015	Mark Rooney	Initial release of FIX engine & documentation
1.0	March 09, 2016	Matthew McAuley	Adding typing functionality within the FIX engine & updating documentation

Contents

Chapter 1

Company Overview

AquaQ Analytics Limited is a provider of specialist data management, data analytics and data mining services. We also provide strategic advice, training and consulting services in the area of market-data collection to clients within the capital markets sector. Our domain knowledge, combined with advanced analytical techniques and expertise in best-of-breed technologies, helps our clients get the most out of their data.

The company is currently focussed on four key areas, all of which are conducted either on client site or near-shore:

- Kdb+ Consulting Services: Development, Training and Support;
- Real Time GUI Development Services;
- SAS Analytics Services;
- Providing IT consultants to investment banks with Java, .NET and Oracle experience.

The company currently has a headcount of 30 consisting of both full time employees and contractors and is actively hiring additional resources. Some of these resources are based full-time on client site while others are involved in remote/near-shore development and support work from our Belfast headquarters. To date we have MSAs in place with 6 major institutions across the UK and the US.

Chapter 2

Overview

This product includes software developed by quickfixengine.org (`http://www.quickfixengine.org/`).

2.1 Features

This FIX engine provides many standard features including:

Easy KDB+ integration - Getting access to a full suite of FIX engine features is as simple as loading a shared library into your KDB+ application.

Supports Standard FIX Versions - The FIX engine supports all of the FIX standards out of the box and even allows you to use multiple different message formats at the same time.

Handling kdb+ types - The adapter handles kdb+ types within messages, converting to FIX string format on sending and back to kdb+ types on message receipt.

Message Validation - All messages that are received by the FIX engine are validated against a specification before being processed to help simplify your business logic.

Custom Message Formats - The engine supports custom specifications via a simple XML format based on the underlying QuickFIX library.

Guaranteed Delivery - Messages are tracked alongside sequence numbers, so if a client misses a message, it can always be recovered.

Runs on Linux & Windows - The adapter is designed so that it will easily build & run on both Windows and Linux.

Chapter 3

System Setup

3.1 Requirements

3.1.1 CMake

This project uses CMake 3.2+ [1] to support building across multiple platforms. The CMake toolchain will generate the build artefacts required for your platform automatically.

3.1.2 KDB+

A recent version of kdb+ (i.e version 3.x) should be used when working with the FIX engine. The free 32-bit version of the software can be downloaded from the Kx Systems website[2].

3.2 Building the Shared Library

The first step is to build the quickfix library itself for the target platform. The instructions for this can be found either on the GitHub page or on the QuickFIX website[3]. You will need to copy the quickfix project to the `third_party/quickfix/<version>` directory for the CMake build to find it. For example, if you were building the project against version 1.14.3 of quickfix, your directory layout should look like the version below:

[1] `http://www.cmake.org/`
[2] `http://kx.com/software-download.php`
[3] `http://www.quickfixengine.org/quickfix/doc/html/building.html`

```sh
$ ls
/home/aquaq/fix-build/third_party/quickfix/1.14.3
total 1.4M
drwxr-xr-x 3 4.0K 2016-01-19 09:31 bin/
drwxr-xr-x 2 4.0K 2016-01-19 09:31 config/
drwxr-xr-x 3 4.0K 2016-01-19 09:31 doc/
drwxr-xr-x 6 4.0K 2016-01-19 09:31 examples/
...
```

This project provides a simple shell script that will handle the build process for you. It will compile all of the artifacts in a /tmp folder and then copy the resulting package into your source directory.You can look at this script for an example of how to run the CMake build process manually.

```sh
./package.sh
```

Figure 3.1: *Creating a build directory and running an out-of-source build with CMake*

A successful build will place a .tar.gz file in the fix-build directory that contains the shared object, a q script to load the shared object into the .fix namespace and some example configuration files.

3.3 Configuration

3.3.1 Adding New Message Schemas

After building and unzipping the package, you should see config folder that sits alongside the shared library. This contains all of the core configuration for both acceptors and initiators. The `spec` folder contains all of the schemas for each version of FIX that is supported.

```q/kdb+
~/fixengine/bin/config/spec> ls FIX*.xml
FIX.4.0.xml
FIX.4.1.xml
FIX.4.2.xml
FIX.4.3.xml
FIX.4.4.xml
FIX.5.0.xml
FIX.5.0.SP1.xml
```

Figure 3.2: *All the FIX message schemas that are supported are defined in the config/spec folder*

q/kdb+

```
/ Creating an acceptor that will validate all of its messages against the FIX.4.2.xml schema file.
/ If a DataDictionary is not specified explicitly, it will be automatically populated by looking for
/ the BeginString.
q) .fix.listen[(enlist `BeginString)!enlist `FIX.4.2]
/ Creating a second acceptor that uses a different schema for validation than the BeginString indicates.
q) .fix.listen[(enlist `DataDictionary)!enlist `$"config/spec/customschema.xml"]
```

Figure 3.3: *Creating fix adapters with default and custom schemas*

3.3.2 Modifying the default configuration file

There are two configuration files in the `config` directory of the FIX engine. The first is `acceptor-config.xml` which contains configuration that is specific to all the acceptors and the `initiator-config.xml` which contains configuration that is specific to all the initiators. Both of these xml files are made up of the same sections `[DEFAULTS]` and `[SESSION]`.

The `DEFAULTS` section of the of the configuration is where you can specify the common settings for all the acceptors that will be launched in the q session.

ini/config

```
# Defaults that should be shared across all sessions
# (they can be overridden on a per-session basis)
[DEFAULT]
BeginString=FIX.4.4
ConnectionType=acceptor
ReconnectInterval=60
SenderCompID=sellside
TargetCompID=buyside1
SocketNodelay=Y
```

Figure 3.4: *Example defaults section of a configuration file*

The `SESSION` sections of the configuration correspond to a specific acceptor that should be launched automatically when the library is loaded. You may have multiple sessions configured as long as they do not share a SessionID or try to bind to the same port. The SessionID is made up of the `BeginString`, `SenderCompID` and `TargetCompID` and the binding port is defined by `SocketAcceptPort`. For initiators, you can have multiple connections to the same SessionID open as long as your specify the SessionQualifier.

ini/config

```
[SESSION]
# Statically creating an acceptor from a configuration file
StartTime=00:30:00
EndTime=23:30:00
ReconnectInterval=30
HeartBtInt=15
TargetCompID=buyside1_44
SocketAcceptPort=7070
SocketReuseAddress=Y
DataDictionary=config/spec/FIX44.xml
AppDataDictionary=config/spec/FIX44.xml
FileStorePath=store
PersistMessages=Y
FileLogPath=logs
```

Figure 3.5: *Defining session specific settings with the [SESSION] tag*

A full listing of all the supported options can be found in the QuickFIX documentation. All of the configuration options that are available in this file are also available at runtime.

3.3.3 Enabling/Disabling Message Verification

Message verification is enabled by default for the shared library. The message validation is carried out by the process that receives the message according to the DataDictionary setting. The data dictionary that is used will be automatically selected based on the contents of the BeginString used when creating an acceptor or an initiator.

If you wish to disable the message verification, you can set the UseDataDictionary variable in the configuration to **N**.

ini/config

```
[DEFAULT]
# To explicitly enable or disable the message verification, we can use the
# UseDataDictionary variable.
#
# e.g UseDataDictionary=N
#
# We can also set a custom message validator by setting the DataDictionary
# variable to point at an XML file with our own rules. If this is set to an
# invalid file path and UseDataDictionary hasn't been disabled, then an error
# will be thrown when an acceptor or initiator is created.
DataDictionary=config/spec/CUSTOM.xml

# It is also possible to override the DataDictionary variable on a per session basis,
# which means you can deal with multiple FIX versions simultaneously within the same
# process (but you cannot deal with multiple FIX versions within a session!).
[SESSION]
DataDictionary=config/spec/FIX.4.4.xml
# ...
[SESSION]
DataDictionary=config/spec/FIX.4.1.xml
```

Figure 3.6: *Configuring message verification via the INI files*

The message verification is only applied when you receive a FIX message and not when you send them. This means that you should check for invalid message format responses from the counter party after you send a message.

3.3.4 Logging

Logging is supported via directing message to stdout, to a flat file, or to a MySQL and PostgreSQL databases. This logging can be configured in the ini files from the previous section or at runtime. If you scroll to the bottom of the configuration file, you should see three sections for each.

For plain text logging to be enabled you just need to define the FileLogPath and FileStorePath variables. If you need to be able to automatically parse and analyse the logging output, it may be better to use either the MySQL or PostgreSQL storage options.

ini/config

```
##############################################
#       Plain Text Logging Configuration
##############################################
FileLogPath=logs
FileStorePath=store
```

Figure 3.7: *Example Plain Text Logging Configuration Settings*

To enable MySQL logging, you just need to enable to `MySQLLogDatabase`, `MySQLLogHost`

and `MySQLLogPort` variables. The default username is root and the default password is empty, so you will mostly likely need to change the `MySQLLogUser` and `MySQLLogPassword` variables if you are running in a production setting.

ini/config

```
##############################################
#       MySQL Logging Configuration
##############################################
MySQLLogDatabase=quickfix
MySQLLogUser=root
MySQLLogPassword=pass
MySQLLogHost=localhost
MySQLLogPort=20017
MySQLLogUserConnectionPool=N
```

Figure 3.8: *Example MySQL Logging Configuration Settings*

The settings for a PostgreSQL database configuration mirror those for the MySQL version. Again, it is possible to split the guaranteed delivery storage from your debug logs by explicitly specifying the variables e.g `PostgreSQLStoreDatabase` alongside `PostgreSQLLogDatabase`.

ini/config

```
##############################################
#       PostgreSQL Logging Configuration
##############################################
PostgreSQLLogDatabase=quickfix
PostgreSQLLogUser=root
PostgreSQLLogPassword=pass
PostgreSQLLogHost=localhost
PostgreSQLLogPort=20017
PostgreSQLLogUseConnectionPool=N
```

Figure 3.9: *Example PostgreSQL Logging Configuration Settings*

3.3.5 Guaranteed Delivery/Message Replay

The FIX engine supports guaranteed delivery and message replay without any extra input from the user. To disable this ability to replay messages, you just need to set the `FileStorePath` variable to be empty. You can also change this variable in order to change the location that these delivery logs are stored. The message replaying can be enabled on a per session basis and it is also possible to store each sessions messages in a different location.

ini/config

```
[DEFAULT]
# Make all acceptors/initiators store their message logs in a store folder.
LogStorePath=store
```

Figure 3.10: *Configuring guaranteed delivery via the INI files*

All of this functionality is integrated into the FIX engine, so you don't need to worry about manually sending requests for missed data. The FIX engine will ensure all updates arrive in the correct order according to the sequence number.

ini/config

```
MySQLStoreDatabase=quickfix
MySQLStoreUser=root
MySQLStorePassword=pass
MySQLStoreHost=localhost
MySQLStorePort=20017
MySQLStoreUserConnectionPool=N
```

Figure 3.11: *Storing message updates in a MySQL database rather than a in plain text*

It is also possible to use a database as a message store by taking the logging examples from above and replacing the 'Log' with 'Store' as show in figure 3.11.

3.4 Start Up

3.4.1 Loading the Shared Library

In order to load the shared library, we will use the dynamic load (2:) function. This function is dyadic and takes a name/path to a library as its first argument and a list as its second argument. The list should contain the name of the function that you wish to dynamically link against and the number of arguments that it takes. The library provides a bootstrapping function: `extern "C" K load_library(K x);`

This bootstrapping function will return a dictionary that maps symbols to dynamically linked functions. The function will erase the contents of any namespace that it is assigned to, so if you want to add additional variables or functions, you will need to place them after the code that loads the shared library.

We have provided a simple fix.q script in the package that will automatically load the library into the .fix namespace for you. It will also provide some useful enumerations for working with fix messages.

q/kdb+

```
/ We load the entire library in one go by loading in the fix.q bootstrapping script.
/ Inspecting the .fix namespace now shows all of the available functions.
q).fix
create                | code
send                  | code
...

/ We can then use the functions in the exact same way that we would use normal
/ kdb+ functions.
```

Figure 3.12: *Loading the library functions by executing the* `load_library` *function*

Once the shared library is loaded, then we can then load the enumerations in the `fixenums.q` file. This will add some dictionaries that are helpful if we want to build the FIX messages by hand.

3.4.2 Starting Servers (Acceptors) and Clients (Initiators)

The `.fix.create` (section 4.1.2) function is common to both starting Acceptors and Initiators.

The Acceptor is the server side component of a FIX engine that provides some sort of service by binding to a port and accepting messages. To start an acceptor you need to call the `.fix.create` function with `` `acceptor `` as the first argument. The second argument may be either an empty list `()` or a specified configuration file `` `:sessions/sample.ini ``. The `.fix.create` function called as an `` `acceptor `` will start a background thread that will receive and validate messages and finally forward them to the `.fix.onrecv` (section 4.1.4) function if the message is well formed.

q/kdb+

```
q) .fix.create[`acceptor;()]
Defaulting to sample.ini config
Creating Acceptor
```

Figure 3.13: *Creating an acceptor that will listen for FIX messages, using the default config file*

The acceptors share a configuration file located at: `sessions/sample.ini`. This file contains some defaults that are to be shared among all of the acceptors that are created if they are not overridden elsewhere. The acceptors that are listed in the configuration file will always be started as soon as the library has been loaded.

ini/config

```
# Create the first session -- remembering to define a unique TargetCompID,
# SenderCompID & SocketAcceptPort.
[Session]
TargetCompID=SessionOneSellerID
SenderCompID=SessionOneBuyerID
SocketAcceptPort=7070
StartTime=00:30:00
EndTime=23:30:00

# Create the second session -- again we need to ensure our SessionID (which is
# in the format BeginString:SenderCompID->TargetCompID) is unique and that we
# have our own port.
[Session]
TargetCompID=SessionTwoBuyerID
SenderCompID=SessionTwoSellerID
SocketAcceptPort=7071
StartTime=05:45:00
EndTime=21:00:00
```

Figure 3.14: *Creating two acceptors by defining them in the acceptor-config.ini file*

The Initiator is the client side component of the FIX engine and is intended to be used to connect to acceptors to send messages. To start an initiator you need to call the `.fix.create` (section 4.1.2) function with `` `initiator `` and a configuration file. This will create a background thread that will open a socket to the target acceptor and allow you to send/receive messages.

q/kdb+

```
q) .fix.create[`initiator;`:sessions/sample.ini]
Creating Initiator
```

Figure 3.15: *Creating an initiator with a specified configuration file*

3.4.3 Sending a FIX Message

In order to send a FIX message from an initiator to an acceptor, you can use the `.fix.send` (section 4.1.3) function. The send is executed synchronously and will either raise a signal upon error, otherwise you can assume that the message has been received successfully by the counter party.

In order to determine who the message will be sent to, the library will read the contents the message dictionary and look for a session on the same process that matches. The BeginString, SenderCompID and TargetCompID fields must be present in every message for this reason.

q/kdb+

```
/ Session 1 - Create an Acceptor
q) .fix.create[`acceptor;()]
Defaulting to sample.ini config
Creating Acceptor

/ Session 2 - Create an Initiator
q) .fix.create[`initiator;()]
Defaulting to sample.ini config
Creating Initiator

/ We can then create a dictionary
/ containing tags and message values
q) message: (8 11 35 46 ...)!("FIX.4.4";175;enlist "D";enlist "8" ...)
/ Then send the message
q) .fix.send[message]
```

Figure 3.16: *Creating an acceptor and an initiator and sending a message between them*

The FIX message itself should just be a dictionary that maps integers (that correspond to the tags defined in the specification) to kdb+ types. The shared library will handle to conversion of the kdb+ type to the format expected by FIX before sending. You can also just pass symbols with the data pre-formatted to the dictionary and the library will pass it straight through to the FIX engine.

Some extra constants are provided to make the building of FIX messages a bit easier. The example below shows how you would typically send a NewOrderSingle("D") message:

q/kdb+

```
.fix.send_new_single_order: {[a;b]
        / The message itself is just a dictionary containing the data that you
        / wish to send to the FIX session. The ordering of the tags doesn't matter
        / as the FIX engine will sort them into the order in the specification before
        / sending the message.
        message:()!()
        message[.fix.tags.BeginString]: "FIX.4.4"
        message[.fix.tags.SenderCompID]: string a;
        message[.fix.tags.TargetCompID]: string b;
        message[.fix.tags.MsgType]: enlist "D";
        message[.fix.tags.ClOrdID]: "SHD2015.04.17"
        message[.fix.tags.Side]: enlist "1";
        message[.fix.tags.TransactTime]: "20150417-17:38:21";
        message[.fix.tags.OrdType]: enlist "1";

        / The .fix.send function will determine where to send the message based
        / on it's contents (e.g SenderCompID & TargetCompID)
        .fix.send[message];
}
```

Figure 3.17: *Example of creating a NewSingleOrder type FIX message and sending it*

3.4.4 Receiving a FIX Message

To receive a FIX message, you will want to use the `.fix.onrecv` callback. This callback should be defined with a single argument: e.g. `.fix.onrecv:{[dict] ... }`. The `.fix.onrecv` callback will be triggered for every *non-administrative* message that is received. You should check the message type field of the FIX version and Message Type of the dictionary which will be always present.

q/kdb+

```
/ Creating a callback that will just print out the dictionary
/ to console.
q) .fix.onrecv{[dict] show dict; }
...
q)
/ Sample output when the callback is triggered
8|  "FIX.4.4"
9|  112
35| "D"
34| 188
49| "SellSideID"
...
```

Figure 3.18: *Setting up the .fix.onrecv handler to print out any messages that are recieved*

You should only rely on the FIX message contents to determine who the message was sent by and to find out which session you should respond to. It is also possible for the message to contain a `` `OnBehalfOf `` tag if the messages are being routed through several different FIX services before they reach you.

3.5 Integration with TorQ

It is common for some tick data to be published as FIX messages and it would be useful to store this data into kdb+ for analysis. The FIX engine can also be integrated with the AquaQ TorQ framework which provides a production ready tick capture environment. More information on TorQ can be found on the resources section of the AquaQ website[4], or on the AquaQ GitHub[5].

The most common scenario would be connecting to a counterparty using an initiator and then using the `.fix.onrecv` function in order to push the data into kdb+. We can set this up by creating a new process in the processes folder of TorQ, loading the shared library, opening a handle to a tickerplant and then watching for Execution Report messages (which have MsgType = 8). Because we receive these messages as a kdb+ dictionary, can easily extract the data that we want to store from the message and then push it to the tickerplant.

[4] `http://www.aquaq.co.uk/products/`
[5] `https://github.com/AquaQAnalytics/TorQ`

Chapter 4

Shared Library API

4.1 Functions

4.1.1 .fix.version

The `.fix.version` function takes no arguments and will return a dictionary containing version information.

q/kdb+

```
.fix.version[]
release | 0.1.3
quickfix| 1.14.3
os      | Linux
kx      | 3
```

Figure 4.1: *Version information for the API and session*

4.1.2 .fix.create

The `.fix.create` function can be used to begin initiator and acceptor sessions.

q/kdb+

```
/ This creates an initiator with the default configuration file.
.fix.create[`initiator;()]
/ We can also create an acceptor, this time with a specified configuration file
.fix.create[`acceptor;`:sessions/sample.ini]
```

Figure 4.2: *Demonstration of how to create an initiator or acceptor using the .fix.create function*

The function takes two arguments, specifying the counterparty side and the config file respectively. The first argument must be either ‘initiator or ‘acceptor and the second can be an empty list, defaulting to the provided sample.ini, or a user specified file.

4.1.3 .fix.send

The `.fix.send` function takes a dictionary that maps long integers to any other kdb+ type. The dictionary should contain all the correct FIX tags and a session associated with the TargetCompID and the SenderCompID should exist. The session should have been created within the same kdb+ instance using `.fix.create` (section 4.1.2). If a session that is associated with the message doesn't exist, then a 'session error will be raised and a more detailed message will be written to stderr.

q/kdb+

```
q) message:(8 35 46 ...)!(`FIX.4.4;`D;`SellerID ...)
q) .fix.send[message]
```

Figure 4.3: *Sending a FIX message by creating a dictionary and filling it with tags.*

The `.fix.send` function will use the BeginString, SenderCompID, and TargetCompID fields in order to lookup which session the message is intended for. This means that these three fields must be set in every message dictionary.

If the message has an invalid header or doesn't contain all the required fields, then a 'config error will be raised in kdb+ and a more detailed error message will be sent to stderr. The validation is performed by the FIX engine that is recieving the message and not locally.

4.1.4 .fix.onrecv

The `.fix.onrecv` function can be defined to take a dictionary and will be called for each non-administrative message that is recieved (i.e. it won't be called for heartbeats, login, logout as these should be automatically handled by the quickfix library). It will typically be called after one of the participants in a session has used `.fix.send` (section 4.1.3).

q/kdb+

```
/ Defining a handler that will just print the dictionary to
/ screen. It will be called each time a non-administrative
/ message is recieved.
q) .fix.onrecv:{[dict] show dict; }
...
8 | FIX.4.4
9 | 112
35| D
34| 250
49| SellSideID
```

Figure 4.4: *Recieving a FIX message using the .fix.onrecv handler. Each FIX field is mapped to a dictionary entry for easy consumption.*

You may add any logic that you want inside the callback function to handle the

dictionary, including sending response messages, disconnecting clients or storing the data in a kdb+ database.

4.2 Enumerations

4.2.1 .fix.tags

The .fix.tags dictionary has been built from the C++ headers in order to make building/decoding FIX messages by hand easier. To load this dictionary just load the fixenums.q script into your session **after** loading the shared library.

q/kdb+

```
q) .fix.tags
               | ::
BeginSeqNo     | 7
BeginString    | 8
BodyLength     | 9
CheckSum       | 10
EndSeqNo       | 16
MsgSeqNum      | 34
MsgType        | 35
...

/ We can look up which integer maps to a specific tag e.g. BeginString
q) .fix.tags.BeginString
8

/ A reverse lookup can also be performed using the find (?)
/ operator as each integer maps to a single tag.
q) .fix.tags?8
`BeginString
```

Figure 4.5: *Using the .fix.tags dictionary to map between the integer form & the human readable representation*

The `fixenums.q` file has been generated by a python script located in `src/python` in the project. To create a new version of the `fixenums.q` file, copy the headers from the `include/quickfix` directory into the same directory as the python script and then run **parseheaders.py**.

4.2.2 .fix.tables

The .fix.tables dictionary is another useful utility mapping alphanumeric message types to their full name.

q/kdb+

```
q).fix.tables
| ::
Heartbeat                | `0
TestRequest              | `1
ResendRequest            | `2
Reject                   | `3
SequenceReset            | `4
Logout                   | `5
IOI                      | `6
...

/ We can look up which alphanumeric maps to a specific message type e.g. BeginString
q) .fix.tables.Heartbeat
`0

/ A reverse lookup can also be performed using the find (?)
/ operator as each alphanumeric maps to a single message type.
q).fix.tables?`0
`Heartbeat
```

Figure 4.6: *Using the .fix.tables dictionary to map between the alphanumeric form & the human readable message type*

