AquaQ Analytics

C++ Feed Handler Whitepaper

email:
support@aquaq.co.uk

web:
`www.aquaq.co.uk`

AquaQ Analytics

Revision History

Revision	Date	Author(s)	Description
0.1	March 30, 2015	Mark Rooney	Initial Draft

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 predominantly 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.

Please feel free to contact us if you feel we may be able to assist you with your kdb+, data or analytics needs.

info@aquaq.co.uk

Chapter 2

Requirements and Resources

In order to follow this document, you should be comfortable with the following topics:

C/C++ It is assumed that the reader will have some intermediate knowledge of C or C++ (e.g. regarding pointers/references).

The KDB+/C API The basics of the C api will generally not be covered in the document. The document will cover some of the issues that are more relevant to writing a feed handler.

IPC in kdb+ You should be able to open connections between multiple q processes and send messages between them. Knowledge of the difference between synchronous and asynchronous communication is also assumed.

kdb+tick You should be familiar with the core components of a basic kdb+tick capture system (e.g. tickerplant, rdb, hdb).

You will need the following tools in order to compile some of the code examples:

Visual Studio 2010+ If you wish to compile the 64 bit versions of the binaries, you will need to have the Ultimate Edition of Visual Studio, or the 2013 Community Edition.

KDB+ 3.2 You can download the latest version of kdb+ from the Kx System website. The code in this document has been tested on version 3.2 (released 2014.12.05).

Some additional documentation that may be useful when following this document:

AquaQ Resources Contains various resources on interfacing with kdb+ via C, TCP and other kdb+ topics that maybe be useful.

Interfacing With C (Kx Wiki) This section of the Kx wiki explains the basics of interfacing with kdb+ via C.

Starting kdb+ Describes the basics of a typical kdb+tick capture system.

Chapter 3

Introduction

The goal of this document is to demonstrate a design for a feed handler that interfaces with Kx Systems kdb+ database [1]. The design allows it to be compiled as both a standalone executable and as a shared library. The standalone executable will run as its own process and communicate with the q processes via IPC. The shared library can be loaded into the q process that wants to collect the data, resulting in lower latency between the feed handler and the receiving process.

The architecture for the feed handler should share as much code as possible in order to increase maintainability. In order to facilitate this, all processing of the data from the feed will take place in a background thread so that the main thread of the q process is still responsive. This use of threading can introduce its own problems when you need to marshal data between the feed handler and the main q process, which we provide a solution for in this paper. The solution provided is just one of the possible ways to structure a feed handler and may not be suitable for more specialized use cases.

To begin, the document will also quickly cover some of the basics of compiling and linking the different types of binaries (executables and libraries) on each platform and explain how to load and use the libraries within a q process. If you don't have a kdb+tick system already set up to test your feed handler, you can download the **AquaQ TorQ**[2] and the **AquaQ TorQ Starter Pack**[3]. TorQ should allow you to get up and running with a production grade kdb+tick setup as soon as possible. The final section of the document will show you how to integrate the feed handler with the TorQ starter pack to provide a realistic use case.

[1] http://kx.com/software.php

[2] https://github.com/AquaQAnalytics/TorQ

[3] https://github.com/AquaQAnalytics/TorQ-Finance-Starter-Pack

Chapter 4

Compiling and Linking

Compiling the feed handler as a shared library and as a standalone executable requires different steps and build artefacts. How you structure your code and build your software can differ based on the platform that your are running on. The example code that is provided alongside this document has been written to compile on both Windows and Linux platforms using the CMake build tool which abstracts some of these details away. We will describe how to build these objects manually for each platform to highlight some of the requirements specific to each. We will indicate any potential cross platform issues as we present code, and examples of how to work around these problems can be found in the example source code.

The first file you will need to download from the Kx subversion repository is the `"k.h"` header[1]. You should include this header whenever you need to interface with kdb+ as it defines all of the types and functions that are necessary. The same header file is used with both version **2.x** and **3.x** of kdb+ and the version is selected by defining `KXVER` to be either **2** or **3**. If `KXVER` is not defined, the compiler should emit an error to notify you of this.

```c
#include <stdlib.h>
#include <stdio.h>

#define KXVER 3 // declaring that we want to work with v3.x of kdb+
#include "k.h"

int main(void)
{
	return EXIT_SUCCESS;
}
```

Figure 4.1: Importing the "k.h" header and defining KXVER in a C program

It is usually possible to define `KXVER` from your build tools when compiling, but this differs depending on the platform and tools. As an example, on Linux with GCC, we could use the **-D** flag to specify `KXVER` instead of using the preprocessor.

[1]http://code.kx.com/svn/kx/kdb+/c/c/k.h

```
gcc example.c -DKXVER=3 -fpic -shared -std=gnu99 -o example.so
```

Figure 4.2: Defining KXVER from the command line with GCC

All of the example code in this document assumes that a more recent version of C is used in order to provide some quality of life improvements in the code (e.g. declaring iteration variables inside the for loop). **Visual Studio** provides a C++ compiler that supports these features out of the box. Standard C compilers such as **GCC** and **clang** will require a flag to enable these features (e.g. *(-std=gnu99)*).

It should be noted that not all functions declared in `"k.h"` are available to both libraries and executables. An example of this is the **sd0** and **sd1** functions which are covered later in the document (it wouldn't make sense for these functions to exist inside a standalone executable).

4.1 Linux

4.1.1 Shared Objects

On Linux, you can create a shared object by simply passing the **-shared** flag when compiling. This will create a **.so** file that exports any functions that are not prefixed with the *static* keyword. It is also important that the binary produced is position independent if it is to be shared between multiple processes, so you should also use the **-fpic** flag when compiling a shared library.

```
        gcc example.c -fpic -shared -std=gnu99 -o example.so
```

Figure 4.3: Compiling a position independent shared library with GCC

You do not need to link against any other objects in order to build or load the library on Linux. To check that the correct functions have been exported, you can use tools such as **objdump** to inspect the binary and view the symbol tables. Passing objdump the **-t** flag will make it print the symbol tables to the console, and you can look for your function name in the rightmost column of the output.

One other issue you will need to take care of is that you want to make sure you are not building a 64 bit shared library to load into a 32 bit q process or vice versa. This can be changed with the **-m32** (for 32 bit) and **-m64** (for 64 bit) flags with GCC. Attempting to load a 64 bit shared object into a 32 bit q process will cause an error to be thrown and the process may exit.

4.1.2 Standalone Executable

The process for creating a standalone executable that can interface with kdb+ is similar to building a shared object. The key difference is that you will need to provide an entry

```
aquaq@localhost:~/fakefeed/bin> objdump -t example.so

example.so:     file format elf64-x86-64

SYMBOL TABLE:
...
0000000000000960 g     F .text  00000000000000d6              GenerateCore
0000000000000b10 g     F .text  000000000000008d              GenerateTrade
0000000000000a40 g     F .text  00000000000000c2              GenerateQuote
...
0000000000000ba0 g     F .text  0000000000000077              ProcessFeed
0000000000000000       F *UND*  0000000000000000              time@@GLIBC_2
    .2.5
00000000000007f8 g     F .init  0000000000000000              _init
```

Figure 4.4: Viewing the symbol table of a shared object using *objdump*

point (i.e. define `int main(...)`) and also link against an object that defines the functions in `"k.h"`. The reason that we didn't need to link against an object for the shared library is that the q process itself will have already loaded them.

On Linux, the **c.o** file contains the implementations and is available in both **32-bit**[2] and **64-bit**[3] versions. It is critical that you load the correct version of c.o to match the executable that you are building.

To build the executable, you just need to pass the **c.o** object alongside your code and make sure that the **-shared** and **-fpic** flags are not defined. You will need to link against the **pthread** library using **-lpthread** too as this is required for some of the functions to execute correctly.

```
gcc example.c c.o -std=gnu99 -lpthread -o example
```

Figure 4.5: Compiling an executable with GCC using the *pthread* library

Similarly to the shared object, you can use the **-m32** and **-m64** flags with GCC in order to build 32-bit or 64-bit executables respectively. Linking against the wrong version of the c.o file can cause runtime errors or segfaults to occur.

4.2 Windows

4.2.1 Shared Objects

On Windows, it is typical to use a tool set such as **Visual Studio**, rather than the commandline tools (e.g **cl.exe**). We will show how to use both Visual Studio projects and the command line in order to compile the code.

[2]http://code.kx.com/svn/kx/kdb+/l32/
[3]http://code.kx.com/svn/kx/kdb+/l64/

In order to build shared objects on Windows, you will need to link against the **q.lib** file which provides the declarations of the exported objects. This is required on Windows, otherwise the functions that are defined in "k.h" will fail to link properly and the DLL file will not load into the q processes. Adding the q.lib file to your project is performed in the same way that you would add the c.obj file. You will also need to make sure you are still linking against the **ws2_32.lib** file. One issue that may arise at runtime is crashing or inconsistent behaviour due to thread local storage (TLS). In cases where your application is incompatible TLS, an alternate implementation of c.obj called **cst.obj** is provided which should solve any issues.

With the Visual Studio toolchain, the functions that are defined are not exported into your library by default. You will need to explicitly list the functions that are to be made available in your library by creating a **.def** file or by using the `__declspec` keywords. The simplest .def file that will work is just a text file with EXPORTS as the first line followed by the names of any functions that you want to export on following lines. The example below will export the *init* and *halt* functions in the library. This file will also prevent any name mangling that would otherwise occur when compiling these functions.

```
EXPORTS
init
halt
```

Figure 4.6: A .def file that tells the windows compiler to make the *init* and *halt* functions available in the symbol table

If you are compiling with the `__declspec` declaration on your functions, the names that are exported will be mangled which makes it difficult (if not impossible) to import these functions into kdb+. To solve this, you will need to prefix your functions with extern "C" to tell the compiler to pass your function names through untouched. It may be helpful to define a macro as below to prefix your functions with to make sure that your API functions are exported correctly.

```
#define FEEDLIBRARY_API extern "C" __declspec(dllexport)

FEEDLIBRARY_API K init(K x);
K reload(K x);
```

Figure 4.7: Explicitly making functions visible using the _ _ *declspec* keyword

Examples of using cl.exe to compile code as a dynamically linked library can be found at: `http://code.kx.com/svn/kx/kdb+/c/c/c.a`

4.2.2 Standalone Executable

On Windows you will need to get the latest c.obj files from http://code.kx.com/svn/kx/kdb+/ in either the **w32** or **w64** folders depending on whether you want to create a 32 bit or 64 bit process. You will also need to compile against the Winsock2 library if running as a standalone executable as this is required by the implementations.

If you have a Visual Studio project set up and do not want to attempt compiling from the command line then you just need to drop the object into the path where Visual Studio can find it. The easiest way to do this is to click on *"Resource Files -> Add... -> Existing Item"* from your solution explorer.

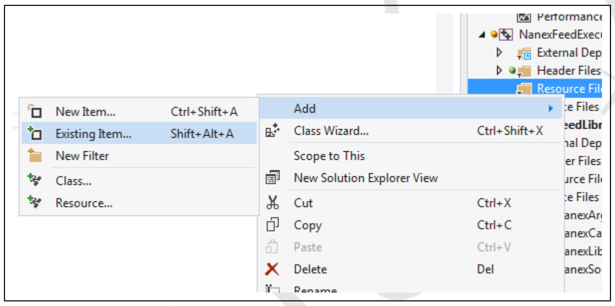
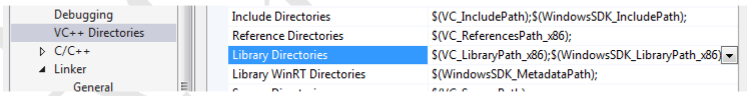


Figure 4.8: Adding an item as a Resource File via the Solution Explorer

Another way to make sure the objects are included by the linker is to add them to the "Input" section of your projects resource page.



Include Directories	$(VC_IncludePath);$(WindowsSDK_IncludePath);
Reference Directories	$(VC_ReferencesPath_x86);
Library Directories	$(VC_LibraryPath_x86);$(WindowsSDK_LibraryPath_x86)
Library WinRT Directories	$(WindowsSDK_MetadataPath);

Figure 4.9: Adding an item as a Resource via the Input Resource Page

Note that we add **WS2_32.lib** to input section of this page to make sure that the Winsock2 library is included. This file is included on the system path of any recent Windows installation (i.e. Windows XP or greater). Once the object files are visible to Visual Studio, you should be able to compile any code without issues.

It is also possible to compile the code from the command line on Windows using **cl.exe**. This can be found by navigating to your Visual Studio installation location:

```
C:\Program Files (x86)\Microsoft Visual Studio\12.0\VC\bin.
```

Figure 4.10: Location of the cl.exe and msbuild.exe tools

This folder should be added to your system PATH, or you can run *vcvars32.bat* from that folder in order to make it visible temporarily from the current session.

Examples of using cl.exe to compile code as a standalone executable can be found at: `http://code.kx.com/svn/kx/kdb+/c/c/c.c`

4.3 Loading shared objects into kdb+

Once you have your shared objects compiled, you should be able to load them into kdb+ using the 2: system call [4]. The example q code below imports three functions: init, find and other from a shared object called example.so. The init and find functions take 0 and 1 arguments respectively. Note that import for the init function declares that it takes 1 argument however! This is because all functions in kdb+ really take at least one argument and that calling a function like init[] is equivalent to init[::].

```
init:`example 2:(`init;1)
find:`example 2:(`find;1)
other:`example 2:(`other;3)
```

Figure 4.11: Importing the functions into kdb from a library called example.so using the dynamic load (**2:**) function.

It is also possible to export these functions from your C code to make it easier to import from kdb+ if you have a large API. You will create a function that returns a dictionary that maps symbols to dynamically loaded functions. An example of such a function is shown below:

```
K load_funcs(K x)
{
        K exportedKeys = ktn(KS, 3);
        K exportedValues = ktn(0, 3);

        kS(exportedKeys)[0] = ss("init");
        kS(exportedKeys)[1] = ss("halt");
        kS(exportedKeys)[2] = ss("get_args");

        kK(exportedValues)[0] = dl(init, 1);
        kK(exportedValues)[1] = dl(halt, 1);
        kK(exportedValues)[2] = dl(get_args, 1);

        return xD(exportedKeys, exportedValues);
}
```

Figure 4.12: A C function that returns a dictionary that maps keys to functions linked using dl

We can then just execute this function from our q script and assign the result to a namespace. *Note that the assignment to the namespace will erase any existing items that were defined in it!*

You should now be able to use the functions from your q script just as you would use any other q function.

[4]http://code.kx.com/wiki/Reference/TwoColon

```
.fh:(`FeedHandlerLibrary 2:(`load_funcs;1))`
```

Figure 4.13: Loading the function *load_funcs*, evaluating it, and then assigning the resulting dictionary to a namespace

4.4 Common Issues

Some common issues that you may run into when loading shared library functions are:

"The symbol is not defined and could not be loaded" - Either the function has not been exported correctly or there is a typo in the code that loads the function.

"Incompatible binary format" - You are trying to load a 32 bit shared library into a 64 bit q process or vice versa.

Chapter 5

Architecture Overview

5.1 Overview

The feed handler will be designed so that as much code as possible is shared between the standalone version and the dynamically linked library version. In this implementation the standalone executable will just parse some arguments, wrap them up into Q/K code and then pass this to the the library init function to start the feed handler. The following figure shows which parts of the code will be shared between the two different implementations.

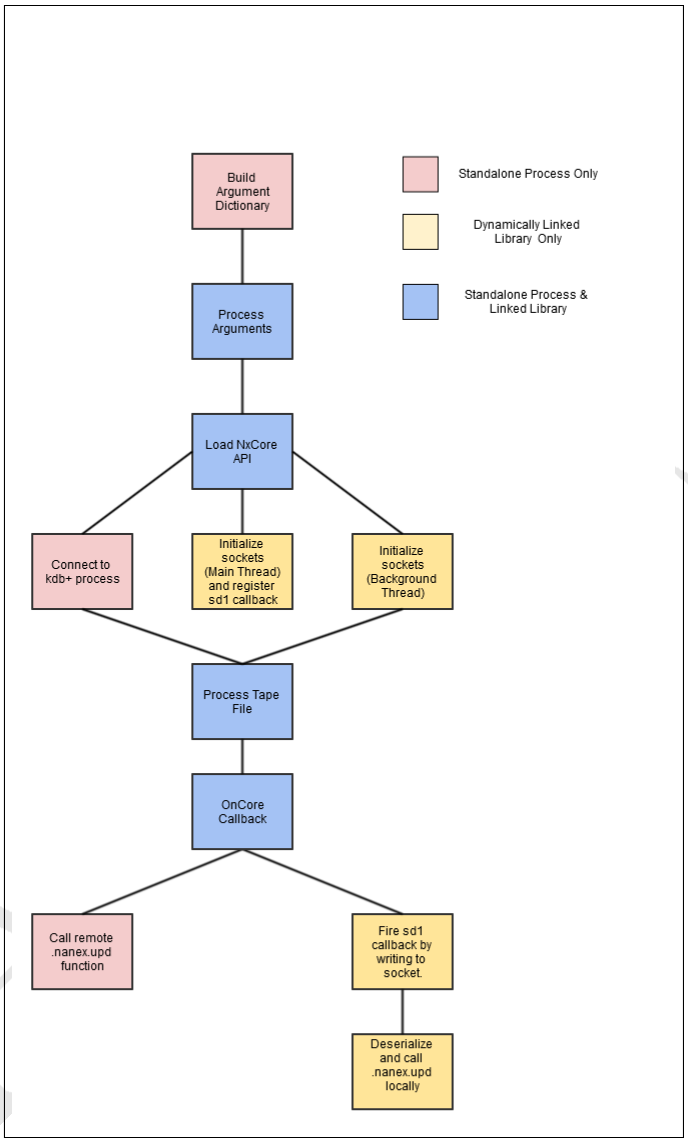

Build Argument Dictionary
Standalone Process Only
Dynamically Linked Library Only
Standalone Process & Linked Library
Process Arguments
Load NxCore API
Connect to kdb+ process
Initialize sockets (Main Thread) and register sd1 callback
Initialize sockets (Background Thread)
Process Tape File
OnCore Callback
Call remote .nanex.upd function
Fire sd1 callback by writing to socket.
Deserialize and call .nanex.upd locally

Figure 5.1: Feed Handler Architecture (both the Shared and Standalone components)

The sections that are highlighted blue are parts of the code base that can be shared between the two different types of feed handler. Most of the code that performs the parsing of the data will be untouched, and the differences are mostly in how the feeds are initialized.

For example, the standalone feed handler is expected to take its arguments from the command line whilst the shared object version takes its arguments as a dictionary passed to the initialization function. The standalone implementation can however re-use the shared objects code by converting it's command line arguments into a K object that can then be passed to the shared object.

In order to keep as much code as possible identical between the two implementations, we will implement the standalone version of the feed handler as a shell that just reads some arguments and then performs it's work via the shared library.

5.2 Output Format

There are some decisions that need to be made in terms of the output that the feed handler will produce. Typically a q process will receive a list of atoms which represent a row to be inserted into the database. The other type of update that a q process will typically handle is a list of column vectors that represents a batch of updates. The feed handler needs to convert a stream of serialized data (e.g. binary/json/xml) into one of these structures before passing it to the receiving process. Batching updates to a q process allows for more efficient IO operations and less TCP/IP overhead. This in practice means that you will be trading off latency for increased bandwidth.

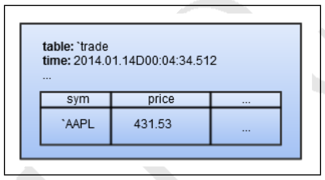


Figure 5.2: Data packet format for the standalone feed handler

The feed handler will output messages in two different forms depending on how it is used. In standalone mode, it will send K objects to the remote process. This allows the receiving q process to decide which fields it wants to store in its tables without re-compiling the feed handler.

When running the feed handler as a dynamically linked library, then we can take the dictionary that is passed to the q process and unpack it before forwarding it onto the q process. This method also allows us to easily add some filter operations before the data is forwarded.

In the provided implementation, this is defined as a .fh.upd function which takes the dictionary data and performs some unpacking and automatic filling of data to match the schema. Through this unpacking process we receive a list of atoms which are then pushed to the q process using the .u.upd function. The dictionary data will be wrapped up inside another dictionary that contains some meta data about the message, such as the time the message was sent and the table that it should be pushed to.

This makes use of a `.fh.tables` dictionary that just maps table names to the column names that are present in that table. The example below shows how this could be implemented. Note that the .fh.createrecord function will drop or fill data as appropriate and also reorder the columns to fit the table schema.

The alternative is to have the feed handler just push a list of atoms directly to the q process which can provide a performance boost but at the cost of flexibility (the feed handler would require compilation each time the schema is changed). This method

```
.fh.tp: .. // this is the handle to the q process
.fh.upd:{[x] .fh.tp(`.u.upd;.fh.createrecord[x`table;x`data]) }
.fh.createrecord:{[t;x] (.fh.tables[t],((cols[.fh.tables[t]] inter key[x])#x))}
```

Figure 5.3: Definitions of *.fh.upd* and *.fh.createrecord* that will unpack the dictionary structure

```
/**
* ProcessFeed is a simple function that populates the statically allocated
* FeedData struct automatically and provides a pointer to it via a callback
* function for each message.
*/
void ProcessFeed(int (*) (const FeedData *));
```

Figure 5.4: The declaration of the callback function ProcessFeed

means that the q process doesn't need to be modified to handle special message types and the feed handler can just be pointed at other kdb+ installs without any issues.

5.3 Use of Background Threads

As we will be running the feed handler as a shared library, it will need to perform most of its work in a background thread so that the q processes is still responsive. The standalone feed handler doesn't need a background thread as it will just be running on its own, however to ensure we share as much code as possible between the two implementations, we will start a background thread and just wait/sleep in the main thread. It is also possible to have variations on this architecture that use more threads to manage different feeds or to spread the work of parsing across different instruments or asset classes for make better use of multiple cores.

In the shared object implementation, the background thread will need to communicate with the main q thread rather than an external q process. There are several possible issues that can pop up related to memory management and concurrency that will be explained later in the document. The solution suggested attempts to avoid concurrency issues by having threads communicate with each other over a socket and minimizes the amount of thought you will need to put into memory management by having each thread only access K objects that it creates.

5.4 Feed API

In order to build the feed handler, we will need an example feed to work with. We will build our own library that has a single function that allows us to subscribe to the contents of the feed. No login or instrument subscription logic will be included in the examples. The function that we use to listen for new trades and quotes is called ProcessFeed.

The function that is passed into ProcessFeed should expect to take a single parameter which is a pointer to a FeedData struct (which holds the trade/quote data) and return an integer which indicates if the feed should continue sending data.

The FeedData data structure is a tagged union. A member called **type** should be checked before accessing the rest of the structure, otherwise you could be reading from invalid data. For example, one you determine that the data received is trade data, you can read the members in **data->core.*** (which are common to both trades and quotes) and also the data in **data->msg.trade.***.

The Feed itself is not intended to replicate a real feed in terms of the number of updates received, or in ensuring that the trades and quotes pair off correctly. It is intended solely to demonstrate connecting to a feed and serializing the data that is received. The FeedData structure is generated within the feed library itself and will be allocated in a static, shared piece of memory. A reference to this structure will then be passed to the feed handler on each callback (this is a common strategy in implementing these types of feeds).

The `feed.so/feed.dll` will be built automatically alongside the code for the feed handler itself, but as a separate object file. The feed itself has no dependencies and is platform independent, so if you would like to use the feed for your own experiments, you should just copy both the `fakefeed.h` header file and the shared object into your own project.

```
typedef struct _CoreMessage {
        char *sym;
        char *exg;
        int sequence;
        char cond[4];
} CoreMessage;

typedef struct _TradeMessage {
        int size;
        int volume;
        double price;
} TradeMessage;

typedef struct _QuoteMessage {
        int asksize;
        int bidsize;
        double askprice;
        double bidprice;
} QuoteMessage;

typedef struct _FeedData {
        int type;
        CoreMessage core;
        union {
                TradeMessage trade;
                QuoteMessage quote;
        } msg;
} FeedData;
```

Figure 5.5: The FeedData structure that is populated by the fake feed

```
int ExtractData(const FeedData *data)
{
        switch(data->type) {
                case FF_TRADE_MSG:
                // trade data can be accessed as in the examples below.
                //
                // data->core.sym;
                // data->msg.trade.size;
                break;
        case FF_QUOTE_MSG:
                // quote data can be accessed as in the examples below.
                //
                // data->core.sym;
                // data->msg.quote.askprice;
                break;
        }
        return FF_CONTINUE_FEED;
}
```

Figure 5.6: Example of processing messages from a feed in the callback

The ProcessFeed function expects a constant `FF_CONTINUE_FEED` to be returned if the feed should continue to process messages, and `FF_HALT_FEED` to be returned if not. These constants are defined in the `fakefeed.h` header file.

5.5 Error Handling

The error handling in the example code is kept to a minimum in order to keep the code simple, however in practice the library code will need to be able to throw errors in both standalone and shared object formats. In order to handle errors in this way, we need to use macro's to conditionally compile some parts of the code. The function should just print an error to stderr when we are running as a standalone executable and then exit. For the shared library, we need to return krr from our function so that it reaches the kdb+ process and raises a signal. This behaviour makes it easy for users to respond to different types of errors in their q scripts without it just killing their process.

The function can then be used in the code by calling it and returning it's result to kdb+.

```
////
// If called inside the standalone executable, this function will print the error to
    stderr and then exit
// immediately without attempting to release resources. If it is called from inside q
     as part of the DLL
// implementation it will just print an error message to the screen and then return.
//
K halt_on_error(const char *lngmsg, const char *kdbmsg)
{
#ifndef FEEDHANDLER_STANDALONE
        return krr(kdbmsg);
#elseif
        print_info(stderr, lngmsg);
        exit(EXIT_FAILURE);
        return (K) 0;
#endif
}
```

Figure 5.7: Definition of halt_on_error that will terminate in a standalone executable and raise a signal in a shared object

```
K init(K x) {
        if (!ProcessArgs(argv, argc)) {
                return halt_on_error("error: argument dictionary is invalid!", "
                    badargs");
        }
}
```

Figure 5.8: Using the halt_on_error function to signal errors in init

Chapter 6

Implementation

This section of the document will show you some of the important steps in implementing an actual feed handler. In order to simplify things, we will focus on the shared library version of the feed handler first and then show the minor modifications to get it running as a standalone executable where appropriate. For a complete example see the code that accompanies this document.

6.1 Initialization

6.1.1 Connecting to a Q Process

The first step you should take in a standalone C program that will be creating and sending K objects is to initialise the kdb+ runtime. This occurs automatically whenever a call to any of the khp* functions are called. If you are not able to open the connection to the q process immediately, then you should still call khp as the very first step in your program but with a port of -1 e.g. khp("",-1). Any code that is running as a shared object should already have the run time initialized correctly by the q process.

Note that a shared library cannot use the khp function to open a connection to another q process and will not compile as these functions are missing from q.lib! If you want to share code between the shared library and the standalone executable, you will need to conditionally compile these parts of the code.

The value returned from the khp call is an integer that represents the connection handle. This handle can now be used to send queries and K objects to a q process. To do this, we use the k function with the first argument set as the handle, the second argument is a string that should be valid q expression and the remaining arguments

```
int conn = khp("localhost", 7010);
if (conn <= 0) { fprintf(stderr, "connection failed!\\n"); exit(EXIT_FAILURE); }
```

Figure 6.1: Opening a connection to a q process with khp

```
k(conn, "a:42", (K) 0);
```

Figure 6.2: Assigning the value 42 to a variable a on a q process

are K objects that should be passed as arguments to the q expression followed by a terminating NULL.

If the handle is negative; then the k() function will operate asynchronously. This simple use of the k() function is how the feed handler will communicate with other processes. One thing to take note of is that all function calls in q take at least 1 argument! If you need to call a function that takes no arguments, you should just pass an integer or a null.

```
k(conn, "function", ki(15), kf(3.142), (K) 0);
```

The standard kdb+ tick.q script has a .u.upd function expects two arguments. The first is a symbol that indicates the table that the data should be stored in. The second argument is the data itself which should be either a list that represents a single row, or a list of column vectors to represent a batch of updates. The time column in either case is optional and it can be appended automatically by the script (which is usually preferable in most setups).

```
k(conn, ".u.upd", ks("trade"), knk(3, ks("IBM"), kf(363.242), kj(2424)));
```

If you are running as part of a shared library, you can still use the k function to send messages to the q process. The shared object call should take a connection parameter of 0 and cannot be called asynchronously.

6.1.2 Exporting an API to kdb+

```
#ifndef FEEDHANDLER_API_H
#define FEEDHANDLER_API_H

#undef UNICODE
#define WIN32_LEAN_AND_MEAN

// âĂę other required include files

#define KXVER 3
#define WIN32 1
#include "k.h"

#define FEEDHANDLER_API extern "C" __declspec(dllexport)

FEEDHANDLER_API K init(K x);
FEEDHANDLER_API K halt(K x);
FEEDHANDLER_API K get_args(K x);

#endif
```

In this example the init function is the way the user will start the feed handler. It will accept arguments as a dictionary. We will also provide a zero-argument halt function that disconnects from the feed and will de-allocate any memory that was used during the last run. The last function get_args will return a dictionary that contains the arguments that were passed to init. The examples below show how this API could be used from q. The library in the examples will be called FeedHandlerLibrary.dll.

```
q) .fh.init:`FeedHandlerLibrary 2:(`init;1) / Load in the init, get_args and halt
    functions
q) .fh.get_args:`FeedHandlerLibrary 2:(`halt;1)
q) .fh.halt:`FeedHandlerLibrary 2:(`get_args;1)
q) .fh.init[(`username`password)!(`$"exampleuser";`$"examplepass")]
[2015:03:20 16:58:01] - Starting Feed Handler process
[2015:03:20 16:58:01] - Setting up connection with tickerplant process:
    127.0.0.1:7010
...
q) .fh.get_args[]
username | `exampleuser
password | `examplepass
q) .fh.halt[]
[2015:03:20 16:58:16] - Stopping Feed Handler
```

6.1.3 Argument Parsing

The first part of the implementation that we will focus on is the argument parsing part of the feed handler. When we are running as a shared object, the init function will be taking a K object as its argument that should be of type dictionary. In our implementation, the init function will immediately pass the dictionary to a function called ProcessArgs which performs type checking of the arguments and parses them into a C struct that will be used throughout the rest of the program. An example of how the init function is structured is listed below:

```
K init(K argsdict)
{
        // ... other pre-argument parsing initialization code

        if (!ProcessArgs(argsdict)) {
                return halt_on_error("unable to process the arguments dictionary!");
        }

        // ... other post-argument parsing initialization code

        return (K) 0;
}
```

The implementation of the ProcessArgs function for a small number of the possible arguments is shown below. You should be careful when processing a dictionary as it can map from almost any kdb+ to any other kdb+ type. This means that you cannot assume that the user will be passing symbols or character vectors as the keys to the dictionary. In this case we will just check that the keys are a list of symbols and then throw an error otherwise.

```
bool ProcessArgs(K x)
{
        // If we were not passed a K object, then we should create an empty
            dictionary.
        if (IsUndefined(x->t)) {
                x = xD(ktn(KS,0),ktn(KS,0));
        }

        // If we get a non dictionary type we should return with an error;
        if (!IsDictionary(x->t)) {
                krr("expecteddict");
                return false;
        }

        K keys = kK(x)[0];
        K values = kK(x)[1];

        for(int i = 0, n = (int) keys->n; i < n; i++) {
                S key = kS(keys)[i];
                K value = kK(values)[i];

                if (0 == strcmp(key, "tapefile")) {
                        SetTapeFile(value);
                } else if (0 == strcmp(key, "stripenum")) {
                        SetStripeNum(value);
                }
        }

        return true;
}
```

The standalone executable will reuse this code from the shared library by taking the arguments from the command line, placing them into a K object and then calling the init function again. This means that we have one central place for configuring the startup logic of the program which makes the application much more maintainable.

```
int main(int argc, char *argv[])
{
        // parse arguments from command line into a settings struct
        // & initialize the q runtime by calling khp before we touch
        // any K objects.

        // Create the vector of keys for the arguments dictionary
        K keys = ktn(KS, 4);
        kS(keys)[0] = ss(âĂIJkdbuserâĂİ);
        kS(keys)[1] = ss(âĂIJkdbpassâĂİ);
        kS(keys)[2] = ss(âĂIJkdbhostâĂİ);
        kS(keys)[3] = ss(âĂIJkdbportâĂİ);

        // Create the mixed list of values for the arguments dictionary
        K values = knk(0, 4);
        kK(values)[0] = ks(settings.user);
        kK(values)[1] = ks(settings.pass);
        kK(values)[2] = ks(settings.host);
        kK(values)[3] = ki(settings.port);

        // Create the dictionary from the keys and values and then call
        // the same init function that would be called directly from the
        // q process.
        init(xD(keys, values));

        return EXIT_SUCCESS;
}
```

6.2 Parsing Data/Serialization

6.2.1 Capturing and Storing Updates

Now that we have some code to initialize our feed handler with some arguments and an API that can be accessed from C, we can now focus on an example of how to parse data from a feed and push it to kdb+.

Different feeds will provide different styles of API for processing the data. In some cases the feed will provide updates one at a time and have the data parsed for you, others will need to be in some raw format and require parsing.

Because the APIs for these feeds differ so much, we will create our own simplified API so that we can focus on creating K objects and pushing them to the q process. Many feeds such as (EbsLive) will provide a socket that can be used in conjunction with the select() call in order to receive updates. Other implementations will hide this communication and instead provide callbacks (through virtual functions or function pointers) that can be registered to specific types of events.

It is also often a requirement to be able to filter the data that is being sent by the feed. Filtering can be performed by having the output configured up stream (i.e. via contract with the feed provider) or by setting flags in the API to indicate that you don't want to receive certain events. If neither of these two options are available then you will need to implement the filtering yourself, either in the feed handler or in the tickerplant part of the code.

We will assume that the Feed API itself is callback/event based via a simplified function called ProcessFeed. It will take a C function as one of its parameters and will repeatedly call this function every time a trading or system event occurs. It will pass a simplified 'FeedData' data structure that contains the message type and the data for the message that is to be parsed into a K object.

```
// We will assume that a function like the one below exists and that it
// will call the fn passed to it each time it gets an event. A real API
// would be more complex to set up and it would involve configuration to
// specify which tables, events, symbols you are interested in.
int ProcessFeed(void (*fn)(const FeedData *data));
```

Our implementation of the callback will then effectively be a switch statement that determines the message type that is being processed by reading the FeedData object and then parses the data into the appropriate types of K objects.

```
#define column(name, type) ktn((type),0)

static inline K CreateTradeSchema()
{
        return knk(6,
                column("symbol", KS),           // symbol
                column("system_time", KP),      // timestamp
                column("price", KF),            // float
                column("size", KJ),                     // long
                column("condition", KH),        // short
                column("sequence", KJ));        // long
}
```

The schema for the objects that are being sent via kdb+ are defined as a mixed list of lists that we will append to. A simple column macro will make the creation of the schema a bit more clear. The first argument to the macro is a string (which is ignored and is just for documentation purposes), and the second is the type of that column in kdb+.

Storing our updates as a mixed list of primitive typed lists allows us to batch updates if required and also to store the data more efficiently than if we sent individual rows to kdb+.

The initial time field for the data is also omitted from the schema, but will be appended to the data automatically once it reaches the tickerplant. When we are performing batched updates however, this means that all items in the same batch will have the same time stamp. For applications that need very accurate timestamps, it may be a good idea to provide another time stamp field that reports the exchange/feed time.

The schema for your table should be matched up with the kdb+ types as close as possible to make sure the data is stored and processed efficiently. A full list of types can be found on the Kx Systems Wiki at: `http://code.kx.com/wiki/Cookbook/InterfacingWithC`

6.2.2 Building K Objects from the Feed Data

With the data structure that will hold our updates defined, we can start to parse the data from the callback into the K object. The main functions that we will use to implement this are the js/ja functions. They allow us to append symbols and other atoms to existing lists. They will automatically reallocate space as needed for the data, so this means that you shouldn't keep any other references to the lists.

```
void ParsingCallback(const FeedData *data)
{
        // This holds the trade data until we are ready to publish it to kdb+.
        static K trades = CreateTradeSchema();

        switch (data->messagetype) {
        case TRADE:
        // We fetch the relevant column from the mixed list by using
        // kK(trades)[x] and then get a reference to that so that we
        // can append new atoms with js/ja.
        js(&kK(trades)[0], ss(data->cstring_symbol);
        ja(&kK(trades)[1], &data->systime);
        ja(&kK(trades)[2], &data->price);
        ja(&kK(trades)[3], &data->size);
        ja(&kK(trades)[4], &data->cond);
        ja(&kK(trades)[5], &data->sequence);
        break;
        // âĂę cases that handle other message types.
        }

        // .. send the trades data to the kdb+ process by communicating over
        // the sockets (this will be explained in the next section).
        if (lastupdatesent > batchtime) SendToKDB(âĂIJtradesâĂİ, trades);
}
```

Because we are storing the data directly into the lists as primitives, we don't need to build any K objects. The only adjustment we need to make to the trade data is to convert the system time sent by the feed.

6.2.3 Serializing the K Objects across a socket

The SendToKDB function performs the serialization of the data and then sends it over the socket. A function called CreatePayload wraps the data up in a dictionary that contains some useful information such as the table name and the message type alongside the data. The implementation of CreatePayload is not shown here, but it just returns a standard dictionary as a K object.

```
void SendToKDB(char *tablename, K data)
{
#ifdef FEEDHANDLER_STANDALONE
        k(FeedHandler.tphandle, âĂIJ.u.updâĂİ, ks(tablename), data, (K) 0);
#else
        static const int BUFFER_SIZE = 8192;
        static char buf[BUFFER_SIZE];

        // This is where we create the K object structure that
        // we want to send to the main thread.
        K payload = CreatePayload(tablename, data);

        // Serialize the K object and then release the original copy
        // so no memory is leaked. Note that we use -1 in this instance
        // as we are running as a shared object in kdb v3.x.
        K bytes = b9(-1, payload);
        r0(payload);

        // Copy the size of the serialized message into front of the buffer
        // followed by the serialized content itself.
        memcpy(buf, (char *) &bytes->n, sizeof(J));
        memcpy(&buf[sizeof(J)], kG(bytes), (size_t) bytes->n);

        // Send the message to the socket and then release the K object that
        // holds the serialized contents.
        send(CLIENT_SOCK, buf, (int) (sizeof(J) + bytes->n), 0);

        // We must clean up the serialized K object as it is not passed to any
        // kdb+ functions that would release it.
        r0(bytes);
#endif
}
```

The implementation of the SendToKDB function is split into two parts for the standalone and shared library. The standalone implementation of SentToKDB is very simple as it just needs to send the data directly to the ticker plant q process. We can just use the k function to do this by calling .u.upd with the table name and the row data that needs to be inserted.

Once the data is sent from the background thread, the main thread will call the ProcessUpdate callback that was registered with sd1 to allow us to de-serialize the data and then forward it onto kdb+.

6.2.4 Memory Management Notes

It is important that you pay attention to the kdb+ reference counting system and the life cycle of your K objects when passing data between threads. You should make use of the **r0** and **r1** functions in order to make sure that objects are not destroyed unexpectedly. Any K objects that are created in a thread **must** be destroyed within the same thread! You should also note that some functions from the C API such as **k** will call r0 on their arguments while others such as **b9** will not. Because of the way kdb+ allocates memory, it can make traditional leak checking tools such as Valigrind[1] less effective. You should make use of tools such as top/free and the .Q.w[] function in kdb+ in order to check for memory leaks (checking the amount used by the process).

In the background threads that we create to do work, we must also call the **m9** function just before they terminate. This is because kdb+ allocates memory on a thread by thread basis, and it needs to know when the thread has completed so it can release the memory allocated to that thread pool.

6.3 Shutting Down

The halt function is the last part of our example Tickerplant API that we need to implement. This function takes 0 arguments and just shuts down all the threads, cleans up the sockets and releases any other memory that would have been allocated during the init function.

```
K halt(K x)
{
        if (!LIBRARY_INITIALIZED) return (K) 0;

#ifndef FEEDHANDLER_STANDALONE
        // For the shared object, we need to make sure that the sd0
        // callback is removed.
        sd0((I)MAIN_THREAD_SOCK);
        // ... code to clean up sockets used for the shared library
        //implementation.
#endif
        // ... code to shut down threads and unload/reset any third
        // party libraries
        return (K) 0;
}
```

Note that we conditionally execute the sd0 code only when we are running as a shared library.

6.4 Unpacking the dictionary format

Depending on the format of the K object that was sent to the tickerplant, we may need to perform some filtering/manipulation of the data once it arrives. In the case where we send a simple list of atoms to the ticker plant or when we send a list of atom vectors

[1]http://valgrind.org/

(for batching), we can just let the tickerplant append a timestamp and log the data before publishing to subscribers.

The dictionary format however needs to be unpacked so that it can be forwarded on to the tickerplant. One way to implement this is to cache the table schemas that are available when the tickerplant is first started. We can use this information to build updates that will match our schemas regardless of what the feed handler sends.

```
// example of the required table data for unpacking the dictionary format
.fh.tables:(tables`.)!{[x] first 0# delete time from `. x} each tables`.
```

Once we have the table schemas, we can extract the data from the feed handler message and call a function to convert this into a valid tickerplant update. We can then send the result straight to the tickerplant.

```
// definition of a function that will take a dictionary update and create a list of
// atoms in the correct order for the tables.
.fh.createrecord:{[t;x] (.fh.tables[t],((cols[.fh.tables[t]] inter key[x]#x))) }
.fh.upd:{[x] .fh.send[x`table;.fh.createrecord[x`table;x`data]]; }
```

It may be convenient to filter messages in the tickerplant as the q code is easy to modify and maintain. For performance however, it is better to filter messages in the feed hander so that you prevent data hitting the network where it is not required.

Chapter 7

Running the Sample Code

Sample code is provided alongside this document to demonstrate the implementation of a shared object feed handler that communicates with a background thread in C. It has been written to run on Linux and Windows using the CMake[1] build tool. It has been tested with version **3.2**, but there should be no issues running the examples with any version **3.x** of kdb+. It should be noted that this code is not production ready as much of the required error handling has been omitted in order to make the code clear.

7.1 Building the Sample Code

You can download the example code from the AquaQ Resources section of the website and extract it to a directory on your. Building on all platforms is initially handled with the CMake build tool (version 2.6+ is required) which will in turn generate the platform specific build files.

After extracting the zip file, the directory structure should something look like the one below. The **src** directory contains the source code for the project and the **CMakeLists.txt** file contains the build instructions for CMake.

The next step is to create a **build**/ directory, switch to that directory and then run `cmake ..`. This will output Makefiles on Linux and Visual Studio solutions on Windows.

[1]http://www.cmake.org/

```
src/                            -- contains the source code for the project
CMakeLists.txt          -- contains instructions on how to build
```

Figure 7.1: Contents of the downloaded zip file

```
aquaq:~/fakefeed> mkdir build
aquaq:~/fakefeed> ls
/home/aquaq/fakefeed
total 16K
drwxr-xr-x 3 4.0K Mar 30 16:25 build/
drwxr-xr-x 2 4.0K Mar 27 13:34 src/
-rw-r--r-- 1  961 Mar 27 13:56 CMakeLists.txt
aquaq:~/fakefeed> cd build
aquaq:~/fakefeed/build> cmake .. && make install
```

Figure 7.2: Building the project on Linux using Makefiles

```
PS C:/Users/AquaQ/FeedHandler> mkdir build
PS C:/Users/AquaQ/FeedHandler/build> cd build
PS C:/Users/AquaQ/FeedHandler/build> msbuild ./ALL_BUILD.vcxproj /p:Configuration=
    Release
PS C:/Users/AquaQ/FeedHandler/build> msbuild ./INSTALL.vcxproj /p:Configuration=
    Release
```

Figure 7.3: Building the Visual Studio projects from the command line

7.1.1 Building on Linux

To finish the build on Linux, run `make install` in order to build all of the binaries. The resulting binaries will be placed in the <**build directory**>/../**bin** folder.

7.1.2 Building on Windows

On Windows, you can either open the Visual Studio solution called `fakefeed.sln` and run the `ALL_BUILD` and `INSTALL` targets. This will place the binaries in the <**build directory**>/../**bin** folder. It is also possible to run these steps manually from the command line using the **msbuild** tool. The msbuild tool will typically not be on your PATH after install visual studio, but a script is provided in `C:\Program Files (x86)\Microsoft Visual Studio\<version>\VC` called `vsvarsall.bat`. This will place all the required tools on your PATH automatically for the duration of command line session.

7.2 Running the Sample Code

Successful execution of the build script should create a **bin** folder with the following contents:

```
        bin/
                feedhandler.so
                fakefeed.so
                run.sh
                run.q
```

```
$ /home/aquaq/fakefeed/bin> ./run.sh
KDB+ 3.2 2014.10.04 Copyright (C) 1993-2014 Kx Systems
l64/ 16()core 24145MB mrooney homer 127.0.0.2 EXPIRE 2015.04.01 aquaq.co.uk #46218

q)trade
time sym exg size volume sequence price cond
--------------------------------------------
q)quote
time sym exg asksize bidsize askprice bidprice sequence cond
------------------------------------------------------------
q)init
code
q).u.upd
{[t;x] t insert raze .z.p,x; }
```

Figure 7.4: Starting the script on Linux and examining the tables/functions

The feedhandler.so file provides the feed handler implementation that will be loaded into the q session using the dynamic load (**2:**) operator. It requires that the fakefeed library is in the same directory to generate trades and quotes before it can send them to kdb+.

The run.q script loads in the functions from the feed handler library and provides the example implementation of .u.upd. This will by default just print out the data that was received to stdout. This script should not be run directly on Linux, but instead launched via `./run.sh`. This will ensure that the *LD_LIBRARY_PATH* environment variable has been set correctly. If you want to run the code without the script, you need to either update your environment variables appropriately, put the libraries on your PATH or place them in the same folder as your q executable. On Windows, the script should just be run directly (i.e.`q run.q`) as it looks in the same directory for shared libraries.

Upon running the ./run.sh script, you will be dropped into a q session with the library already loaded. There will also be definitions of two tables, *quote* and *trade* that will hold the data while the feed handler is running. You should be able to see three functions: *init*, *halt* and *.u.upd*. Running the init function will start the feed handler and allows it to generate random trade and quote data using the fake feed library. The .u.upd function will be called once for each update (which could be either a single update or a batch of messages).

You can then start the feed handler by running the `init` function which will cause quotes and trades to start appearing in their respective tables. You can run the halt function to stop the feed handler, however this will not clear the current table data in the q process.

To stop the feed handler, just run the halt function which will cause the feed handler to stop all the background threads and release all unused memory that it had allocated.

```
q)init[]
q)count trade
22411
q)count quote
467599
q)halt[]
q)10#trade
q)10#trade
time                          sym  exg  size volume sequence price cond
-----------------------------------------------------------------------------
2015.03.30D15:29:51.186145000 GOOG BARK 334  990    102596   182.9 0x20494946
2015.03.30D15:29:51.186312000 GOOG CME  328  11     82843    417.3 0x49202043
2015.03.30D15:29:51.186514000 APPL BARK 360  739    37111    298.9 0x43432046
2015.03.30D15:29:51.186819000 MSFT CME  324  341    92989    177   0x4043205a
2015.03.30D15:29:51.186834000 APPL LSE  218  561    4307     381   0x20494043
2015.03.30D15:29:51.186863000 GOOG BARK 340  607    154825   197.7 0x4340435a
2015.03.30D15:29:51.187741000 IBM  CME  18   255    32858    201.1 0x46494349
2015.03.30D15:29:51.187848000 MSFT LSE  169  829    65794    413.8 0x205a4946
2015.03.30D15:29:51.187899000 YHOO BARK 285  864    185921   494.3 0x5a49495a
2015.03.30D15:29:51.188097000 IBM  LSE  401  380    101563   304.3 0x20494920
```

Figure 7.5: Starting and stopping the feed handler and viewing the received trades & quotes

Chapter 8

Conclusion

This document has covered the design and implementation of a typical feed handler in C or C++ using the Kx C API. We have demonstrated the basics of how to communicate with kdb+ using handles and K objects, using threads to perform work in the background and pushing data from a background thread to the main q thread so updates can be performed.