**The Transfer of Data between kdb+ and C**

**AquaQ Analytics Limited**



**Authors**

This document was prepared by:

|  |  |  |
| --- | --- | --- |
| Kent Lee (primary author) |  |  |

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version Number** | **Revision Date**  **dd/mm/yyyy** | **Summary of Changes** | **Document Author** |
| 1.0  1.1  1.2 | 21/02/2013  21/03/2013  13/03/2015 | Initial Release  First Revision  Fixing Memory Leaks & Added cross platform build | See above list  Kevin Piar  Mark Rooney |

**Table of Contents**

[1 Company Background 4](#_Toc350257735)

[2 Overview 5](#_Toc350257736)

[3 Requirements 6](#_Toc350257737)

[4 Data Transfer 7](#_Toc350257738)

[4.1 From kdb+ to C 9](#_Toc350257739)

[4.2 From C to kdb+ 10](#_Toc350257740)

[5 Code Walkthrough: From kdb+ to C 11](#_Toc350257741)

[5.1 Printing an atom 12](#_Toc350257742)

[5.2 Printing a list 16](#_Toc350257743)

[5.3 Printing a table 19](#_Toc350257744)

[5.4 Printing a dictionary 21](#_Toc350257745)

[6 Code Walkthrough: From C to kdb+ 22](#_Toc350257746)

[6.1 Creating an atom 23](#_Toc350257747)

[6.2 Creating a list 28](#_Toc350257748)

[6.3 Creating a dictionary 30](#_Toc350257749)

[6.4 Creating a table 31](#_Toc350257750)

[7 References 32](#_Toc350257752)

# Company Background

AquaQ Analytics Limited (www.aquaq.co.uk) is a provider of specialist data management, data analytics and data mining services to clients operating within the capital markets and financial services sectors. Based in Belfast, the company was set up in April 2011. 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 three key areas:

* Kdb+ Consulting Services – Development, Training and Support (both onsite and offsite).
* Real Time GUI Development Services (both onsite and offsite).
* SAS Analytics Services (both onsite and offsite).

For more information on the company, please email us on info@aquaq.co.uk.

# Overview

Similar data types found in programming languages can have major or subtle differences. Therefore when two interfacing two languages it is important to properly handle the translation of data between each language, for both correctness and efficiency.

This documentation will examine how to transfer several of the kdb+ data types to C and then using C, print the data on the screen. It will also explore the process of creating standard C data types and then returning them to kdb+ for further use.

The code samples associated with this document and the content of the document itself are provided as is, without any guarantees or warranty. Although the author has attempted to find and correct any bugs in the code and in the accompanying documentation, the author is not responsible for any damage or losses of any kind caused by the use or misuse of the code or the material presented in the document. The author is under no obligation to provide support, service, corrections, or upgrades to the code and accompanying documentation.

# Requirements

1. Kdb+ 3.2 (32-bit) [Release date: 2014.12.05]
2. Linux/Unix, Windows, Mac OSX (32-bit)
3. CMake 2.6+
4. Visual Studio Community Edition (only for Windows)
5. makeprint.zip

# Data Transfer

**Compiling shared object**

This project uses CMake in order to compile on Linux, Windows and Mac OSX platforms with little difference in the build steps. On Linux, Windows and Mac OSX, just create a build directory and run “cmake ..” from inside it. This will generate the rest of the build files that are specific to your system.



The output of the build process on all platforms is a shared object that can be dynamically loaded into kdb+ and a script that will load this object for you. You can run the example by typing “q makeprint.q” from the bin directory once the build has finished.

In order to complete the build on Windows, we just need to run the two Visual Studio projects that were generated by cmake and then move to the ../bin/ directory.



On Linux and Mac OSX, we just need to run “make install” to complete the build.



Once the installation has finished, you can just run the makeprint.q script and experiment with the functions!



Please note that header file “k.h” from the code.kx.com repository is required for compiling the C code. To obtain or examine this header file, visit:

<http://code.kx.com/svn/code/kx/kdb+/c/c/k.h>

Before the “k.h” header file can be used, KXVER should be defined in the C code. A value of 3 should be used in order to be compatible with the kdb+ 3.x data structures and a value of 2 to be compatible with version 2.x.

It can either be defined in the C code with the #define KXVER 3 (as in our code samples) or it can be defined from the command line: ‘gcc –DKXVER=3 makeq.c’

**Q script to transfer data**



The C functions can now be dynamically loaded from the shared object into a q session (See <http://code.kx.com/wiki/Reference/TwoColon> for further information). This example assumes that the shared object is located in the current working directory.

## Transferring data from kdb+ to C

To transfer data which has been created in a q session to C and to print to screen, use the “printq” function. The argument to the function is any simple q type (atom, list, table and dictionary). Some simple examples are shown below.

For an atom:



For a list:



For a table:



For a dictionary:



## Transferring data from C to kdb+

To create kdb+ datatypes from the shared C library, use the make function. The argument of the function is a character that represents the type of data to be created. Simple examples are shown below.

(See <http://code.kx.com/wiki/Reference/Datatypes> for further information)

**For a Time atom:**



**For a Time vector:**



**For a Table:**



**For a Dictionary:**



# Code Walkthrough: kdb+ to C

This section explains how to extract the data from a K object and display it in a suitable format.

Before printing the data, the C code will check whether the object is an atom, a list, a table or a dictionary. This can be achieved by inspecting the *t* member of the K object. A macro is available called xt that expands to x->t in the “k.h” header file. This maybe be used in some of the examples.

If the value of x->t is negative then the data is an atom or an error type (-128). If the value of x->t is between 0 and 19, then the data is a list. If it is 98, then the data is a table and if it is 99, the data is a dictionary. Other types that may be received are enumerated types (20 to 76), nested types (77 to 87) and function types (100 to 112). These types are not supported by the printing code and will return a ‘notimplemented signal when passed as a parameter.

In keeping with standard kdb+ notation if the type is negative, the data is an atom. If it is between 0 and 19, the data is a list. If it is 98, the data is a table and if it is 99, the data is a dictionary. However enumerated types between 20 and 76, nested types between 77 and 87 and function types of 100 to 112 are not supported.

More information on types can be found here:

<http://code.kx.com/wiki/Reference/Datatypes>



## Printing an atom

In the event of an atom being passed in the C function call, the first task is identifying what type of atom it is. Each type of atom is accessed differently but all follow a similar pattern. Further information on the C printf format identifiers used can be found at

<http://www.cplusplus.com/reference/cstdio/printf/>.

**Boolean**

A boolean atom is accessed by inspecting the *g* member of the K object.



**Byte**

A byte atom is accessed by inspecting the *g* member of the K object.



**Short**

A short atom is accessed by inspecting the *h* member of the K object.



**Int**

An integer atom is accessed by inspecting the *i* member of the K object.



**Long**

A long atom is accessed by inspecting the *j* member of the K object.



**Real**

A real atom is accessed by inspecting the *e* member of the K object.



**Float**

A float atom is accessed by inspecting the *f* member of the K object.



**Char**

A char atom is accessed by inspecting the *i* member of the K object. This is because char is stored as ASCII code. (i.e. character “a” is stored as 97 according to ASCII code)



**Symbol**

A symbol atom is accessed by inspecting the *s* member of the K object.



**Timestamp**

A timestamp atom is accessed by inspecting the *j* member of the K object.



Timestamp data is stored in nanoseconds in kdb+. Scaling adjustments are needed since time is stored in seconds in C. In addition the epoch defined within kdb+ is 2000.01.01T00:00:00 compared to that of C which is 1970.01.01T00:00:00.

There are 8.64e13 nanoseconds in a day. Therefore the accessed atom is converted to days by dividing by this number. Next, 10957 days are added since there are 10957 days difference between 1970 and 2000. The result is then multiplied by 8.64e4 because there are 8.64e4 seconds in a day.

See <http://www.cplusplus.com/reference/ctime/tm/> for more information on the time data structure in C.

**Month**

A month atom is accessed by inspecting the *i* member of the K object. A month atom is stored as number of months since 2000.01 in kdb+. A simple calculation can be carried out in order to print a month atom in the proper format.



**Date**

A date atom is accessed by inspecting the *i* member of the K object. Another slight adjustment is needed as date is stored as the number of months since 2000.01.01 in kdb+.



**Datetime**

A datetime atom is accessed by inspecting the *f* member of the K object. This is because a datetime is stored as the number of days since 2000.01.01 in kdb+. The fractional part of the number represents the portion of current day elapsed.



**Timespan**

A timespan atom is accessed by inspecting the *j* member of the K object. This is because a timestamp is stored in nanoseconds in kdb+.



**Minute**

A minute atom is accessed by inspecting the *i* member of the K object. Another slight adjustment is needed since minute data is stored in minutes in kdb+.



**Second**

A second atom is accessed by inspecting the *i* member of the K object.



**Time**

A time atom is accessed by inspecting the *i* member of the K object. Again, a slight adjustment is needed since time is stored in milliseconds in kdb+.



## Printing a list

The C code will check what type of list it’s going to print and pass each element in the list to the “printatom” function. In most cases the data in the list will not be represented as K objects and each must be accessed with the correct accessor. The printatoms macro shown below will help eliminate this repeated pattern throughout the code and also make sure that the K objects are allocated correctly.

The first parameter is the K object constructor function (e.g ki, kb etc…). The second parameter is the accessor function (e.g. kI, kG). The third parameter is the K object that you wish to iterate.



For example printatoms(kb, kG, x) would expand into the following code:



**Mixed**

A general mixed list is accessed using the *kK* function. The type of each element in the mixed list is checked to make sure it is an atom instead of a list.[[1]](#footnote-2)



**Boolean**

A boolean list is accessed using *kG* function. The element is then encoded into an atom of the K format using the *kb* function. It is now passed to the “printatom” function.



**Byte**

A byte list is accessed using the kG function and converted to a kdb+ byte with the kg function.



**Short**

A short list is accessed using the *kH* function. The element is now encoded into an atom of the K format using the *kh* function. It is then passed to the “printatom” function.



**Integer**

A short list is accessed using the *kH* function. The element is now encoded into an atom of the K format using the *kh* function. It is then passed to the “printatom” function.



**Long**

A short list is accessed using the *kH* function. The element is now encoded into an atom of the K format using the *kh* function. It is then passed to the “printatom” function.



**Real**

A short list is accessed using the *kH* function. The element is now encoded into an atom of the K format using the *kh* function. It is then passed to the “printatom” function.



**Float**

A short list is accessed using the *kH* function. The element is now encoded into an atom of the K format using the *kh* function. It is then passed to the “printatom” function.



**Char**

A char list is accessed using the *kC* function. The element is now encoded into an atom of the K format using the *kc* function and then passed to the “printatom” function.



**Timestamp**

A timestamp list is accessed using the *kJ* function. The element is now encoded into an atom of the K format using the ktj(-KP,x)function and then passed to the “printatom” function.



**Month**

A month list is accessed using the *kI* function. The element is now encoded into an atom of the K format by creating an empty atom using the *ka(-KM)* function and then passed to the “printatom” function.



**Timespan**

A timespan list is accessed using the *kJ* function. The element is now encoded into an atom of the K format using the *ktj(-KN,x)* function and then passed to the “printatom” function.



**Second**

A second list is accessed using the kI function. The element is now encoded into an atom of the K format by creating an empty atom using the ka(-KV) function and then passed to the printatom function.



## Printing a table

First, the code will ensure that a keyed table is unkeyed using the *ktd* function. This will have no effect on an unkeyed table.



To access the column names and the data within, the table is converted into a dictionary. This can be achieved by inspecting the *k* member of the K object. A dictionary is formed by 2 objects, keys and values. Hence the keys are a list of column names and the values are lists of data.

The list of column names is accessed using kK(x->k)[0] and the data is accessed using kK(x->k)[1] from the flipped table.



The code will then print the table, starting with the column names. The column names are accessed using *kS* function as they are symbols.

The data accessed using kK(x->k)[1] is an array of arrays. So each column is accessed using kK(data)[col]. The printitem function is used to generically print the values in the column regardless of the type. This means that nested and mixed types are supported within the tables without any issues.



## Printing a dictionary

As mentioned before, a dictionary is formed by 2 objects, keys and values. Hence, the keys can be accessed using kK(x)[0] and the values can be accessed using kK(x)[1]. The keys list and the value list are guaranteed to be of equal length.



The code will first print the key, followed by the value associated with that key. For dictionaries within this example, the value can be either an atom or a list. The printitem function is used to print the value in each row as it will handle mixed lists by recursively calling printq.



The printitem function is defined as shown below. For atomic types, it converts the data into a K object, prints the atom and then releases the K object. For complex mixed list types, it will call printq to make sure that these are handled correctly.



# Code Walkthrough: From C to kdb+

This section explains how data created in C is then transferred to kdb+ for further use. The C data in these examples is generated randomly.

The source code will create and return a data type that depends on the argument passed to it from a q session. The argument is a symbol which, if a single lower case letter, represents an atom type. An upper case letter represents a list while `dictionary, `table and `nest represent a dictionary and a table respectively.



## Creating an atom

The code will first establish what type of atom is to be generated. Each type of atom is created differently and examples of their varying creation can be seen below.

**Boolean**

A boolean atom is created using the kb function. Since a boolean can only hold either 0 or 1 as values, a random number between 0 and 1 is generated using rand()%2 where *%*(modulus) retrieves the remainder of the division.



**Byte**

A byte atom is created using the kg function.



**Short**

A short atom is created using the kh function.



**Int**

An int atom is created using the ki function.



**Long**

A long atom is created using the kj function.



**Real**

A real atom is created using the ke function.



**Float**

A float atom is created using the kf function.



**Char**

A char atom is created using the kc function.



**Symbol**

A symbol atom is created using the ks function and a random string generator.



The random symbol generator takes a buffer and a buffer size as parameters and it will fill the first size – 1 characters of the buffer with random letters from the alpha array. This function will guarantee that a terminating null will be written at the end of the string.



**Timestamp**

A timestamp atom is created using the ktj(-KP,x)function. The date and time are represented as the number of nanoseconds since the kdb+ epoch.



**Month**

A month is created using the ka(-KM) function to create an empty month atom. The i member is then set to be the number of months since the kdb+ epoch.



**Date**

A date atom is created using the kd function. The parameter is an integer that represents the number of days since the kdb+ epoch. The function ymd will convert a year, month and day triple into the number of days since the epoch.



**Datetime**

A datetime atom is created using the *kz* function. The argument to this function is a floating point number that is built as described in the logic below.

Note that this type is deprecated. If possible, the more accurate timespan type should be used instead!



**Timespan**

A timespan atom is created using the ktj(-KN,x) function. A timespan is more accurate than a datetime and has its arguments specified as a 64 bit long that is the number of nano seconds since the kdb+ epoch.



**Minute**

A minute atom is created using the *ka(-KU)* function. The *i* field of the ka function should be set to the number of seconds since the kdb+ epoch.



**Second**

A second atom is created using the ka(-KV) function. A second is then specified using variables, as seen below. After which it’s converted into seconds using another simple calculation.



**Time**

A time atom is created using the kt function. The time is then specified using variables, as seen below. After which it’s converted into milliseconds by preforming another simple calculation.



## Creating a list

Once again the code will first establish what type of list is to be generated. Each type of list is created differently and a selection of examples can be seen below.

Lists that are not mixed are usually more efficiently packed and are not stored using the K object structure. This can make initializing the elements in C a little bit more difficult and repetitive. To help with this a simple macro called makeatoms will be defined. The macro will expand into a boiler plate for-loop that helps initialize an array with some expression.



**Mixed**

A general mixed list is created using the ktn(0,n) function and a selection of random atoms are inserted into the list. The atoms are created using the make atom function and a character selected from the list of types. Note that this will only create mixed lists of atoms and will never nest the types.



**Boolean**

A boolean list is created using the *ktn(KB,n)*. Booleans are not stored as a list of K objects in the same way that the mixed list is. They are instead more efficiently packed as bytes. We use the makeatoms macro with arguments makeatoms(KB, kG, rand() % 2).



**Int**

An integer list is created using the ktn(KI,n) function and integer atoms are inserted into the list.



**Symbol**

A symbol list is created using the *ktn(KS,n)* function and symbol type atoms are inserted into the list.



## Creating a dictionary

As discussed above, a dictionary has 2 important components. These are the keys and values. The number of keys must be exactly equal to the number of values in the dictionary.

The key of dictionary is a list of symbols. To create a list of symbols, it is necessary to intern strings using the ss function before storing them into a symbol vector.



## Creating a table

A table is simply a flipped version of dictionary. Hence a table can be created by first creating a dictionary and flipping the result using the xT function. This will only work in the case where the items in the dictionary are all lists of the same length! This is the case in the makedic function, but you should be careful to ensure that this invariant is maintained in the C code!



# References

For more information about interfacing and extending with C, please refer to the following links:

<http://code.kx.com/wiki/Cookbook/InterfacingWithC>

<http://code.kx.com/wiki/Cookbook/ExtendingWithC>

1. Currently the code only supports atoms in a mixed list. This can be extended by the reader if desired. Note that more efficient methods to display vector data may be available. [↑](#footnote-ref-2)