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	21/02/2013	Initial Release	See above list
1.1	21/03/2013	First Revision	Kevin Piar
1.2	13/03/2015	Fixing Memory Leaks & Added cross platform build	Mark Rooney

Table of Contents

1 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.

2 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.

3 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)

4 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.

```
mkdir build
cd build
cmake ..
```

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.

```
msbuild ALL_BUILD.vcxproj /p:Configuration=Release
msbuild INSTALL.vcxproj /p:Configuration=Release
cd ../bin
```

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

```
make install && cd ../bin
```

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

```
C:/Users/AquaQ/qtoc/bin> q makeprint.q
q) makeq["x"]
0xa3
q) printq[00:30:24]
00:30:24
```

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

```
//load in the function
printq:`:./all 2:(`printq;1)
makeq:`:./all 2:(`makeq;1)

//create a table with all types
tab:flip ((`$"t_",/:"mdz")!{x$3?9}'["mdz"]),f:(`$"t_",/:lower[p])!{x$string[3?9]}each
p:except[trim distinct upper[.Q.t];"MDZ"]

update t_mix:(12;1b;2012.03m) from `tab

-1"To Print:\ni.e. printq[1b] for atom\ni.e. printq[101b] for list\ni.e. printq[flip tab]
for dictionary\ni.e. printq[tab] for table"

-1"To Make:\n\"bghijefcspmdznuvt\" for atom\n\"BGHIJEFCSPMDZNUVT0\" for
list\n\"dictionary\" for dictionary\n\"table\" for table\n\"nest\" for nested table\ni.e.
makeq[\"b\"]"

\f
```

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.

4.1 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:

```
q) print[1]
1
```

For a list:

```
q) print[1 2 3]
1 2 3
```

For a table:

```
q) print[([] a:1 2 3)]
a
-
1
2
3
```

For a dictionary:

```
q) print[`a`b`c!1 "a" 3.14]
a | 1
b | "a"
c | 3.14
```

4.2 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:

```
q) make[`t]
07:46:57.915
```

For a Time vector:

```
q) make[`T]
07:46:57.115  13:41:57.939  21:86:52.516
```

For a Table:

```
q) make[`table]
t_b t_x t_h t_i t_j t_e   t_f   t_c t_s t_p                          t_m    ..
-------------------------------------------------------------------------..
1   00  78  64  81  4.91  48.27 s   gG  2017.07.16D18:09:52.000025667 2014.11..
1   45  58  5   27  29.95 54.36 C   Bw  2019.08.11D23:51:02.000030333 2017.10..
0   18  62  45  61  19.42 23.91 x   Kf  2013.09.22D07:53:28.000025547 2019.12..
```

For a Dictionary:

```
q) make[`dictionary]
t_b| 0                             0                             0          ..
t_x| 1e                            0d                            44         ..
t_h| 0                             91                            62         ..
t_i| 55                            10                            59         ..
t_j| 24                            37                            48         ..
t_e| 64.83                         75.95                         40.41      ..
t_f| 36.02                         43.5                          2.91       ..
t_c| A                             O                             w          ..
t_s| Ux                            wH                            Ms         ..
t_p| 2011.07.26D07:18:38.000006900 2018.08.20D08:13:48.000022483 2017.10.03D1..
t_m| 2016.12                       2011.09                       2019.08    ..
t_d| 2016.11.08                    2014.05.21                    2019.02.27 ..
t_z| 2015.02.14T15:43:27.314       2013.09.01T18:18:00.796       2018.06.18T1..
t_n| 2D10:52:38.000024179          0D17:38:11.000019815          8D04:51:02.0..
t_u| 32:15                         48:26                         22:26      ..
t_v| 11:53:49                      06:24:36                      17:18:02   ..
t_t| 03:01:59.557                  12:52:29.075                  08:20:50.003..
t_0| 2011.01.18D21:15:43.000005002 2012.02.13                    2017.10m   ..
```

5 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

```
K printq(K x)
{
    K result;

    if (xt < 0) result = printatom(x);
    else if ((xt >= 0) && (xt < 20)) result = printlist(x);
    else if (xt == 98) result = printtable(x);
    else if (xt == 99) result = printdict(x);
    else result = krr("notimplemented");

    return result;
}
```

5.1 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.

```
case  -1: snprintf(buffer, 4096, "%db", x->g); break;
```

Byte

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

```
case  -4: snprintf(buffer, 4096, "0x%02x", x->g); break;
```

Short

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

```
case  -5: snprintf(buffer, 4096, "%d", x->h); break;
```

Int

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

```
case  -6: snprintf(buffer, 4096, "%d", x->i); break;
```

Long

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

```
case  -7: snprintf(buffer, 4096, "%lld", x->j); break;
```

Real

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

```
case  -8: snprintf(buffer, 4096, "%f", x->e); break;
```

Float

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

```
case  -9: snprintf(buffer, 4096, "%f", x->f); break; // float
```

Char

A char atom is accessed by inspecting the *g* member of the K object. This is because the character is stored as a single byte. (i.e. character "a" is stored as 97 in ASCII)

```
case -10: snprintf(buffer, 4096, "\"%c\"", x->i); break;
```

Symbol

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

```
case -11: snprintf(buffer, 4096, "`%s", x->s); break;
```

Timestamp

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

```
case -12: fmt_time(buffer, 4096, "%Y.%m.%dD%H:%M:%S.",
                      (time_t)((x->j)/8.64e13+10957)*8.64e4); break;
```

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 days since 2000.01 in kdb+. A simple calculation can be carried out in order to print a month atom in the proper format.

```
case -13: snprintf(buffer, 4096, "%04d.%02d", (x->i)/12+2000, (x->i)%12+1); break;
```

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+.

```
case -14: fmt_time(buffer, 4096, "%Y.%m.%d", ((x->i)+10957)*8.64e4); break;
```

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.

```
case -15: fmt_time(buffer, 4096, "%Y.%m.%dD%H:%M:%S", ((x->f)+10957)*8.64e4); break;
```

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+.

```
case -16: { int pos = fmt_time(buffer, 4096, "%jD%H:%M:%S", (x->j) / 1000000000, 1);
            snprintf(buffer + pos, 4096 - pos, ".%09lld", (x->j) % 1000000000); break; }
```

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+.

```
case -17: fmt_time(buffer, 4096, "%H:%M", (x->i) * 60, 1); break;
```

Second

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

```
case -18: fmt_time(buffer, 4096, "%H:%M:%S", x->i, 1); break;
```

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+.

```
case -19: { int pos = fmt_time(buffer, 4096, "%H:%M:%S", (x->i) / 1000, 1);
            snprintf(buffer + pos, 4096 - pos, ".%03d", (x->i) % 1000); break; }
```

5.2 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.

```
#define printatoms(c,a,x) do { \
    int i; \
    for (i = 0; i < x->n; i++) { \
        K r = c(a((x))[i]); \
        printatom(r); \
        r0(r); \
    } \
    } while(0)
```

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

```
int i;
for (i = 0; i < x->n; i++) {
       K r = kb(kG(x)[i]);
       printatom(r);
       r0(r);
}
```

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]

```
case 0: printq(kK(x)[index]); break;
```

[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.

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.

```
case  1: printatoms(kb, kG, x); break;
```

Byte

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

```
case  4: printatoms(kg, kG, x); break;
```

Short

A short list is accessed using the *kH* function and converted into a kdb+ short with the kh function.

```
case  5: printatoms(kh, kH, x); break;
```

Integer

A short list is accessed using the *kH* function and converted into a kdb+ integer with the *ki* function.

```
case  4: printatoms(ki, kI, x); break;
```

Long

A short list is accessed using the *kH* function and converted into a kdb+ long with the *kj* function.

```
case  4: printatoms(kj, kJ, x); break;
```

Real

A short list is accessed using the *kH* function and converted into a kdb+real using the *ke* function.

```
case  8: printatoms(ke, kE, x); break;
```

Float

A short list is accessed using the *kH* function and converted into a kdb+ float using the *ke* function.

```
case  9: printatoms(kf, kF, x); break;
```

Char

A character list is accessed using the *kC* function and converted into a kdb+ char using the kc function.

```
case  10: printatoms(kc, kC, x); break;
```

Timestamp

A timestamp list is accessed using the *kJ* function and converted to a kdb+ timestamp using the ktj function. The first argument to ktj should be *-KP*.

```
//timestamp   i.e. dateDtimespan x2
case(12):{    K obj = ktj(-KP,kJ(x)[i]);
              printatom(obj);
              r0(obj);
              }break;
```

Month

A month list is accessed using the *kI* function and converted to a kdb+ month by creating a month atom with ka and then initializing it using the *i* member of the object.

```
//month              i.e. 2010.01 2010.02
case(13):{    K obj = ka(-KM);
              obj->i = kI(x)[i];
              printatom(obj);
              r0(obj);
              }break;
```

Timespan

A timespan list is accessed using the *kJ* function and converted to a kdb+ timespan using the *ktj* function. The first argument to *ktj* should be *–KN*.

```
//timespan    i.e. 0D00:00:00.000000000 x2
case(16):{    K obj = ktj(-KN,kJ(x)[i]);
              printatom(obj);
              r0(obj);
              }break;
```

Second

A second list is accessed using the `kI` function and converted to a kdb+ second by creating a second atom with ka. It is then initialized using the I member of the object.

```
//second      i.e. 00:00:00 00:00:01
case(18):{    K obj = ka(-KV);
              obj->i = kI(x)[i];
              printatom(obj);
              r0(obj);
              }break;
```

5.3 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.

```
K flip = ktd(x);
```

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.

```
K colName = kK(flip->k)[0];
K colData = kK(flip->k)[1];
```

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.

```
static K printtable(K x)
{
   // Convert the table to an unkeyed one (tables that are already unkeyed
   // are not affected).
   K flip = ktd(x);

   // Get references to the column and row lists and gather their counts.
   // To get the number of rows, we inspect the first columns size.
   K columns = kK(flip->k)[0];
   K rows = kK(flip->k)[1];

   int colcount = columns->n;
   int rowcount = kK(rows)[0]->n;

   // Print out the table headers based on the column names.
   for (int i = 0; i < colcount; i++)
       printf("%s\t", kS(columns)[i]);

   // Print out the table data using the printitem function to handle any
   // nested types.
   for (int i = 0; i < rowcount; i++)
       for (int j = 0; j < colcount; j++)
          printitem(kK(rows)[j], i);

   return (K) 0;
}
```

5.4 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.

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

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`.

```
for (int row = 0; row < keys->n; row++) {
      printf("%s\t| ", kS(keys)[row]);
      printitem(data, row);
}
```

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.

```
#define showatom(c,a,x,i) do { K r = c(a((x))[i]); printatom(r); r0(r); } while(0)

static K printitem(K x, int index)
{
    switch (xt) {
        case  0: printq(kK(x)[index]); break;
        case  1: showatom(kb, kG, x, index); break;
        case  4: showatom(kg, kG, x, index); break;
        case  5: showatom(kh, kH, x, index); break;
        case  6: showatom(ki, kI, x, index); break;
        case  7: showatom(kj, kJ, x, index); break;
        case  8: showatom(ke, kE, x, index); break;
        case  9: showatom(kf, kF, x, index); break;
        case 10: showatom(kc, kC, x, index); break;
        case 11: showatom(ks, kS, x, index); break;
        case 14: showatom(kd, kI, x, index); break;
        case 15: showatom(kz, kF, x, index); break;
        default: return krr("notimplemented");
    }

    return (K) 0;
}
```

6 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.

```
K makeq(K x)
{
    if (xt != -11) return krr("type");

    if (0 == strcmp(x->s, "dictionary")) return makedict(x);
    else if (0 == strcmp(x->s, "table")) return xT(makedict(x));
    else if (strlen(x->s) != 1) return krr("notimplemented");

    if (isupper(x->s[0])) return makelist(x->s[0]);
    else return makeatom(x->s[0]);
}
```

6.1 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.

```
case 'b': return kb(rand() % 2);
```

Byte

A byte atom is created using the `kg` function.

```
case 'x': return kg(rand());
```

Short

A short atom is created using the `kh` function.

```
case 'h': return kh(rand() % 100);
```

Int

An int atom is created using the `ki` function.

```
case 'i': return ki(rand() % 100);
```

Long

A long atom is created using the `kj` function.

```
case 'j': return kj(rand() % 100);
```

Real

A real atom is created using the ke function.

```
case 'e': return ke(rand() / 100.0);
```

Float

A float atom is created using the kf function.

```
case 'f': return kf(rand() / 100.0);
```

Char

A char atom is created using the kc function.

```
#define charfrom(x) x[rand()%(sizeof((x)-1))]
static const char alpha[] = “abcdefghijklmnopqrstuvwxyz”;

case 'c': return kc(charfrom(alpha));
```

Symbol

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

```
case 's': return ks(gensym(buffer, rand() % 32));
```

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.

```
static char* gensym(char *buffer, size_t size)
{
	int i;
	for (i = 0; i < size - 1; i++)
		buffer[i] = charfrom(alpha);
	buffer[i] = '\0';

	return buffer;
 }
```

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.

```
case 'p': {
            long long year = rand() % 10 + 2010;        // year 2010 to 2019
            long long month = rand() % 12 + 1;          // month 1 to 12
            long long day = rand() % 28 + 1;            // day 1 to 28
            long long hour = rand() % 24;               // hour 0 to 23
            long long min = rand() % 60;                // minute 0 to 59
            long long sec = rand() % 60;                // second 0 to 59
            long long nano = rand() % 1000000000;       //nanosecond 0 to 999999999
            long long tq =
                    (((ymd(year,month,day)*24+hour)*60+min)*60+sec)*1000000000+nano;
            return ktj(-KP, tq); }
```

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 days since the kdb+ epoch.

```
case 'm': {
            long long year = rand() % 10 + 2010;        // year 2010 to 2019
            long long month = rand() % 12 + 1;          // month 1 to 12
            K result = ka(-KM);
            result->i = (year-2000)*12+month-1;
            return result; }
```

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.

```
case 'd': {
       long long year = rand() % 10 + 2010;      // year 2010 to 2019
       long long month = rand() % 12 + 1;        // month 1 to 12
       long long day = rand() % 28 + 1;          // day 1 to 28
       return kd(ymd(year, month, day)); }
```

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!

```
case 'z': {
            double year = rand() % 10 + 2010;    // year 2010 to 2019
            double month = rand() % 12 + 1;      // month 1 to 12
            double day = rand() % 24;            // day 1 to 28
            double hour = rand() % 24;           // hour 0 to 23
            double min = rand() % 60;            // minute 0 to 59
            double sec = rand() % 60;            // second 0 to 59
            double mili = rand() % 1000;         // millisecond 0 to 999
            return kz(ymd(year, month, day)+(hour+(min+(sec+mili/1000)/60)/60)/24); }
```

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.

```
case 'n': {
            long long day = rand() % 10;                // day 0 to 9
            long long hour = rand() % 24;               // hour 0 to 23
            long long min = rand() % 60;                // minute 0 to 59
            long long sec = rand() % 60;                // second 0 to 59
            long long nano = rand() % 1000000000;       // nanosecond 0 to 999999999
            return ktj(-KN, (((day*24+hour)*60+min)*60+sec)*1000000000+nano); }
```

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.

```
case 'u': {
            int min = rand() % 60;        // minute 0 to 59
            int sec = rand() % 60;        // second 0 to 59
            K result = ka(-KU);
            result->i = min * 60 + sec;
            return result; }
```

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.

```
case 'v': {
            int hour = rand() % 24;       // hour 0 to 23
            int min = rand() % 60;        // minute 0 to 59
            int sec = rand() % 60;        // second 0 to 59
            K result = ka(-KV);
            result->i = (hour*60+min)*60+sec;
            return result; }
```

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.

```
case 't': {
            int hour = rand() % 24;       // hour 0 to 23
            int min = rand() % 60;        // minute 0 to 59
            int sec = rand() % 60;        // second 0 to 59
            int mili = rand() % 1000;     // millisecond 0 to 999
            return kt(((hour*60+min)*60+sec)*1000+mili); }
```

6.2 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.

```
#define makeatoms(t,a,v) do { \
    result = ktn(t,3); \
    for (i = 0; i < 3; i++) { \
        a(result)[i] = (v); \
    } \
    } while (0)
```

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.

```
//mixed
case '0':{    char typec[]="bghijefcspmdznuvt";
              result = ktn(0,lenq);
              for(i=0;i<lenq;i++){
                     K c = kc((int)typec[rand()%(sizeof(typec)-1)]);
                     kK(result)[i] = makeatom(c);
                     r0(c);

              };
              }break;
```

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).

```
case 'B': makeatoms(KB, kG, rand() % 2); break;
```

Int

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

```
case 'I': makeatoms(KI, kI, rand() % 100); break;
```

Symbol

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

```
case 'S': makeatoms(KS, kS, gensym(buffer, rand() % 32)); break;
```

6.3 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.

```
static K makedict(K x)
{
    static char buffer[] = "t_ ";
    int ntypes = arraylen(types);

    // Note that the keys and values are vectors of the same length.
    K keys = ktn(KS, ntypes);
    K values = ktn(0, ntypes);

    for (int i = 0; i < ntypes; i++) {
        // update the buffer containing the type name and then create
        // a symbol using it. The string needs to be interned with ss
        // before it is assigned into kS(keys)[i] otherwise kdb+ could
        // segfault.
        buffer[2] = types[i];
        kS(keys)[i] = ss(buffer);

        // build the value (we create lists of length 3 so that this can
        // be easily converted into a table using the flip/xT functions)
        kK(values)[i] = makelist(types[i]);
    }

    // Create the dictionary by passing the two vectors to xD.
    return xD(keys, values);
}
```

6.4 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!

```
return xT(makedict(x));
```

7 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