Transfer Different Types of Data Between Kdb+ and C

AquaQ Analytics Limited

Authors

This document was prepared by:

Kent Lee (primary author) Kevin Piar Ronan Pairceir		

Revision History

Version Number	Revision Date dd/mm/yyyy	Summary of Changes	Document Author
1.0	21/02/2013	Initial Release	See list above

Table of Contents

1 Company Background 4

2 Document Overview 5

3 Requirements for running sample code 6

4 Data Transfer between kdb+ and the C Programming Language 7

4.1 Compiling the shared object 7

4.2 Q script to transfer data 8

4.3 Commands for transferring data from kdb+ to C 8

4.4 Transferring data from the C Programming Language to kdb+ 9

5 Code Walkthrough: kdb+ to C 11

5.1 Printing an atom 11

5.2 Printing a list 16

5.3 Printing a table 18

5.4 Printing a dictionary 19

6 C Code Walkthrough: C to kdb+ 20

6.1 Creating an atom 21

6.2 Creating a list 27

6.3 Creating a dictionary 29

6.4 Creating a table 29

6.5 Creating a nested table 30

7 References 31

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 Document Overview

Data types can differ in major or more subtle ways across different programming languages. It is important when transferring data between different languages, to ensure the data is converted to the most appropriate data type in order to ensure correctness and efficiency.

This documentation will examine how to transfer data from kdb+ data types to the appropriate data types in C and then also using C, print the data to 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 for running sample code

1) Trial version of kdb+ 2.8 (32-bit) [Release date: 2013.01.04]

2) Linux/Unix (32-bit)

3) GCC compiler [version 4.6.3]

4) makeprint.zip containing the sample code

4 Data Transfer between kdb+ and the C Programming Language

4.1 Compiling the shared object

A simple makefile (or batch file for windows based systems) is included to compile the c code and produce a shared object to be dynamically loaded by kdb+:

```
all.so:printq.c makeq.c
	gcc -shared printq.c makeq.c -o all.so
```

Please note that header file "k.h" is needed for kdb+ to interact with C.

KXVER is defined in the C code, therefore there is no need to use –D for compiling in the command line. This definition is important as it ensures the correct K structure are used. To obtain or examine this header file, visit:

http://mrbook.org/tutorials/make/.

To compile the C source files, type "make" in a UNIX session from the appropriate directory. To compile the files in a Windows session, use a developer command prompt and enter "make.bat" from the appropriate directory.

```
wooikent@wooikent-VirtualBox:~/C_Prac/makeprint$ make
gcc -shared printq.c makeq.c -o all.so
```

4.2 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). This example assumes that the shared object is located in the current working directory.

4.3 Commands for transferring data from kdb+ to C

To transfer data which has been created in a Q session to C and print it to screen, use the "printq" function. The argument of the function is the data to be deserialized within C and displayed.. Simple examples are shown below.

For an atom:

```
q)printq[1]
Int(-6) 1
```

For a list:

```
q)printq[1 2 3]
Int(-6) 1       Int(-6) 2       Int(-6) 3
```

For a table:

```
q)printq[([]a:1 2 3)]
Table(98)
a
Int(-6) 1
Int(-6) 2
Int(-6) 3
```

For a dictionary:

```
q)printq[(enlist `a)!enlist(1 2 3)]
Dictionary(99)
a       | Int(-6) 1     Int(-6) 2       Int(-6) 3
```

4.4 Transferring data from the C Programming Language to kdb+

To transfer data which has been created in C to kdb+ using appropriate data types, use the "makeq" 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)makeq["t"]
07:46:57.915
```

For a Time Vector:

```
q)makeq["T"]
17:55:46.492 09:01:02.027 02:19:23.926
```

For a Table:

```
q)makeq["table"]
t_b t_x t_h t_i t_j t_e   t_f   t_c t_s t_p                          t_m    ..
-------------------------------------------------------------------------..
1   5c  14  59  40  57.36 25.67 S   Xr  2019.04.02D16:31:22.131176229 2013.03..
1   31  -25 63  26  52.11 64.29 C   Jm  2013.10.08D08:37:38.973594324 2010.05..
0   15  82  26  72  53.68 57.82 D   OW  2015.07.18D22:31:20.911759956 2011.10..
```

For a Dictionary:

```
q)makeq["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   ..
```

For a Nested Table:

```
q)makeq["nest"]
t_B  t_X      t_H         t_I      t_J      t_E              t_F            ..
-------------------------------------------------------------------------..
101b 0x425428 57  -80 -37 9  36 10 21 55 19 56.67 17.05 62.28 24.22 72.69 13...
110b 0x5a544c -42 39  64  42 87 6  99 21 4  11.27 91.5  59.84 40.81 56.3  0.8..
010b 0x2a2407 76  40  27  1  13 72 39 11 40 66.58 39.2  92.24 92.92 19.72 76...
```

5 Code Walkthrough: kdb+ to C

This section explains how a K object is decoded into basic C objects and printed on screen 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. 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){

	//Check for atom
	if(x->t<0){printatom(x);printf("\n");}

	//Check for list
	else if(x->t>=0 && x->t<=19){printlist(x);}

	//Check for table
	else if(x->t==98){printtab(x);}

	//Check for dictionary
	else if(x->t==99){printdic(x);}

	else{printf("cannot support %d type\n",x->t);}

	return (K)0;

}
```

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.

```
//boolean       i.e. 0b
case(-1):{    printf("Boolean(%d) %d",typeq,x->g);
              }break;
```

Byte

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

```
//byte          i.e. 0x00
case(-4):{    printf("Byte(%d) %02x",typeq,x->g);
              }break;
```

Short

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

```
//short         i.e. 0h
case(-5):{    printf("Short(%d) %d",typeq,x->h);
              }break;
```

Int

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

```
//int           i.e. 0
case(-6):{    printf("Int(%d) %d",typeq,x->i);
              }break;
```

Long

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

```
//long          i.e. 0j
case(-7):{    printf("Long(%d) %lld",typeq,x->j);
              }break;
```

Real

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

```
//real          i.e. 0e
case(-8):{    printf("Real(%d) %f",typeq,x->e);
              }break;
```

Float

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

```
//float          i.e. 0.0
case(-9):{     printf("Float(%d) %f",typeq,x->f);
               }break;
```

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)

```
//char           i.e. "a"
case(-10):{    printf("Char(%d) %c",typeq,x->i);
               }break;
```

Symbol

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

```
//symbol         i.e. `a
case(-11):{    printf("Symbol(%d) %s",typeq,x->s);
               }break;
```

Timestamp

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

```
//timestamp      i.e. dateDtimespan
case(-12):{    time_t timval=((x->j)/8.64e13+10957)*8.64e4;
               struct tm *timinfo;
               timinfo=localtime(&timval);
               printf("Timestamp(%d) %04d.%02d.%02dD%02d:%02d:%02d.%09lld",typeq,timinfo-
>tm_year+1900,timinfo->tm_mon+1,timinfo->tm_mday,timinfo->tm_hour,timinfo-
>tm_min,timinfo->tm_sec,(x->j)%1000000000);
               }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 information on time 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.

```
//month          i.e. 2000.01m
case(-13):{    int timy=(x->i)/12+2000;
               int timm=(x->i)%12+1;
               printf("Month(%d) %04d.%02d",typeq,timy,timm);
               }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+.

```
//date           i.e. 2000.01.01
case(-14):{    time_t timval=((x->i)+10957)*8.64e4;
               struct tm *timinfo;
               timinfo=localtime(&timval);
               printf("Date(%d) %04d.%02d.%02d",typeq,timinfo->tm_year+1900,timinfo-
>tm_mon+1,timinfo->tm_mday);
               }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.

```
//datetime     i.e. 2012.01.01T12:12:12.000
case(-15):{    time_t timval=((x->f)+10957)*8.64e4;
               struct tm *timinfo;
               timinfo=localtime(&timval);
               printf("Datetime(%d) %04d.%02d.%02dT%02d:%02d:%02d.%03lld",typeq,timinfo-
>tm_year+1900,timinfo->tm_mon+1,timinfo->tm_mday,timinfo->tm_hour,timinfo-
>tm_min,timinfo->tm_sec,(long long)(round((x->f)*8.64e7))%1000);
               }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+.

```
//timespan      i.e. 0D00:00:00.000000000
case(-16):{   time_t timval=(x->j)/1000000000;
              struct tm *timinfo=localtime(&timval);
              printf("Timespan(%d) %dD%02d:%02d:%02d.%09lld",typeq,timinfo-
>tm_yday,timinfo->tm_hour-1,timinfo->tm_min,timinfo->tm_sec,(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+.

```
//minute        i.e. 00:00
case(-17):{   time_t timval=(x->i)*60;
              struct tm *timinfo=localtime(&timval);
              printf("Minute(%d) %02d:%02d",typeq,timinfo->tm_hour-1,timinfo->tm_min);
              }break;
```

Second

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

```
//second        i.e. 00:00:00
case(-18):{   time_t timval=(x->i);
              struct tm *timinfo=localtime(&timval);
              printf("Second(%d) %02d:%02d:%02d",typeq,timinfo->tm_hour-1,timinfo-
>tm_min,timinfo->tm_sec);
              }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+.

```
//time          i.e. 00:00:00.000
case(-19):{   time_t timval=(x->i)/1000;
              struct tm *timinfo=localtime(&timval);
              printf("Time(%d) %02d:%02d:%02d.%03d",typeq,timinfo->tm_hour-1,timinfo-
>tm_min,timinfo->tm_sec,(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. The data in each type of list is obtained using the associated accessor function. A selection of examples can be seen below:

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]

```
//mixed             i.e. 1 1b 0x01
case(0):{     if(kK(x)[i]->t<0){printatom(kK(x)[i]);}
              else{printf("cannot support list in mixed list\n");return(K)0;}
              }break;
```

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.

```
//boolean       i.e. 01b
case(1):{     K obj = kb(kG(x)[i]);
              printatom(obj);
              }break;
```

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.

```
//short             i.e. 1 2h
case(5):{     K obj = kh(kH(x)[i]);
              printatom(obj);
              }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.

Char

v

```
//char          i.e. "ab"
case(10):{      K obj = kc(kC(x)[i]);
                printatom(obj);
                }break;
```

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.

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

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.

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

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.

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

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.

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

```
for(col=0;col<nCols;col++){

      if(col>0)printf("\t");
      printf("%s",kS(colName)[col]);

}
```

The data accessed using *kK(x->k)[1]* is an array of arrays. So each column is accessed using *kK(data)[col]*. Each list of column data is examined for its type to ensure the code is printing the correct type of data.

```
K list = kK(colData)[col];
if(col>0)printf("\t");
switch(list->t){

	//mixed
	case(0):{	if(kK(list)[row]->t<0){printatom(kK(list)[row]);}
			else{printf("cannot support list in mixed list\n");return(K)0;}
			}break;

	//boolean
	case(1):{	K obj = kb(kG(list)[row]);
			printatom(obj);
			}break;
…
…
…
```

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].*

```
K keyName = kK(x)[0];
K keyData = kK(x)[1];
```

The code will first print the key, followed by the values. For dictionaries within this example, the value can be either an atom or a list.

```
for(row=0;row<nName;row++){

	printf("%s\t| ",kS(keyName)[row]);

	if(kK(keyData)[row]->n>1){printlist(kK(keyData)[row]);}
	else{printatom(kK(keyData)[row]);printf("\n");}

}
```

6 C Code Walkthrough: 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 a data type that depends on the argument passed to it from a q session. After which it returns the newly created data to the same q session. The argument is a character which, if lower case, represents a type of atom. An upper case character represents a type of list while "dictionary", "table" and "nest" represent a dictionary, table and nested table respectively.

```
K makeq(K x){

	//Check the input whether it is a character
	if( (x->t!=10) && (x->t!=-10) ){krr("character");return (K)0;};

	K result;

	switch((char)x->i){

		case 'b':case 'g':case 'h':case 'i':case 'j':case 'e':case 'f':case 'c':
		case 's':case 'p':case 'm':case 'd':case 'z':case 'n':case 'u':case 'v':
		case 't':{	result=makeatom(x);
				}break;

		case '0':case 'B':case 'G':case 'H':case 'I':case 'J':case 'E':case
'F':case 'C':
		case 'S':case 'P':case 'M':case 'D':case 'Z':case 'N':case 'U':case 'V':
		case 'T':{	result=makelist(x);
				}break;

		default:{	char buffer[100];
				int i;

				//Put the object into a buffer
				for(i=0;i<x->n;++i){buffer[i]=(char)kC(x)[i];}buffer[i]=0;

				if( strcmp(buffer,"dictionary") == 0 ){result=makedic(x);}

				else if( strcmp(buffer,"table") == 0 ){result=maketab(x);}

				else if( strcmp(buffer,"nest") == 0 ){result=makenest(x);}

				else{	printf("cannot make %s type\n",buffer);
					return (K)0;}

				}break;
	}
	return result;
}
```

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.

```
//boolean
case 'b':{    result = kb(rand()%2);
              }break;
```

Byte

A byte atom is created using the *kg* function. The random value ranges from 0 to 99.

```
//byte
case 'g':{    result = kg(rand()%100);
              }break;
```

Short

A short atom is created using the *kh* function. The random value ranges from 0 to 99.

```
//short
case 'h':{    result = kh((short)rand()%100);
              }break;
```

Int

An int atom is created using the *ki* function. The random value ranges from 0 to 99.

```
//int
case 'i':{    result = ki((int)rand()%100);
              }break;
```

Long

A long atom is created using the *kj* function. The random value ranges from 0 to 99.

```
//long
case 'j':{    result = kj((long)(rand()%100));
              }break;
```

Real

A real atom is created using the *ke* function. The random value ranges from 0 to 99.

```
//real
case 'e':{    result = ke((float)(rand()%10000/100.0));
              }break;
```

Float

A float atom is created using the *kf* function. The random value ranges from 0 to 99.

```
//float
case 'f':{    result = kf((double)(rand()%10000/100.0));
              }break;
```

Char

A char atom is created using the *kc* function. The character is randomly selected from an array of characters.

```
//char
case 'c':{    static const char charq[]=
                     "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                     "abcdefghijklmnopqrstuvwxyz";
              result = kc(charq[rand()%(sizeof(charq)-1)]);
              }break;
```

Symbol

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

```
//symbol
case 's':{    char buffer;
              gen_random(&buffer);
              result = ks(ss(&buffer));
              }break;
```

The random string generator will select 2 random characters from an array of characters specified in the array *alpha[]*.

```
void gen_random(char *s){

      int i;
      static const char alpha[]=
            "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            "abcdefghijklmnopqrstuvwxyz";

      //Generate 2 random characters and put them in s
      for(i=0;i<2;++i){
            s[i]=alpha[rand()%(sizeof(alpha)-1)];
      }
      s[i]=0;
}
```

Timestamp

A timestamp atom is created using the *ktj(-KP,x)* function. The date and timespan are then specified using variables, as seen below. After which it's converted into nanoseconds using a simple calculation. Note that *ymd()* is used to convert a date into days.

```
//timestamp
case 'p':{    long long yearq = rand()&10+2010,  //year 2010 to 2019
              monthq = rand()%12+1,              //month 1 to 12
              dayq = rand()%28+1,                //day 1 to 28
              hourq = rand()%24,                 //hour 0 to 23
              minq = rand()%60,                  //minute 0 to 59
              secq = rand()%60,                  //second 0 to 59
              nanoq = rand()%1000000000,         //nanosecond 0 to 999999999
              tq  =  (((ymd(yearq,monthq,dayq)*24+hourq)*60+minq)*60+secq)*1000000000+nanoq
              result = ktj(-KP,tq);
              }break;
```

Month

A month atom is created using the *ka(-KM)* function. The month is then specified using variables, as seen below. After which it's converted into months using a simple calculation.

```
//month
case 'm':{    int yearq = rand()%10+2010, //year 2010 to 2019
              monthq = rand()%12+1;       //month 1 to 12
              result = ka(-KM);
              result->i = (yearq-2000)*12+monthq-1;
              }break;
```

Date

A date atom is created using the *kd* function. The date is then specified using variables, as seen below. After which it's converted into days using *ymd*.

```
//date
case 'd':{    int yearq = rand()%10+2010, //year 2010 to 2019
              monthq = rand()%12+1,       //month 1 to 12
              dayq = rand()%28+1;         //day 1 to 28
              result = kd(ymd(yearq,monthq,dayq));
              }break;
```

Datetime

A datetime atom is created using the *kz* function. The date and time are then specified using variables, as seen below. After which it's converted into days using another simple calculation. Please note that division is used with this data type since kdb+ stores datetime in float format.

```
//datetime
case 'z':{    double yearq = rand()%10+2010,     //year 2010 to 2019
              monthq = rand()%12+1,              //month 1 to 12
              dayq = rand()%28+1,                //day 1 to 28
              hourq = rand()%24,                 //hour 0 to 23
              minq = rand()%60,                  //minute 0 to 59
              secq = rand()%60,                  //second 0 to 59
              miliq = rand()%1000,               //millisecond 0 to 999
              tq = ymd(yearq,monthq,dayq)+(hourq+(monthq+(secq+miliq/1000)/60)/60)/24;
              result = kz(tq);
              }break;
```

Timespan

A timespan atom is created using the *ktj(-KN,x)* function. The day and time are specified using variables, as seen below. After which it's converted into nanoseconds using some simple manipulation.

```
//timespan
case 'n':{    long long dayq = rand()%10, //day 0 to 9
              hourq = rand()%24,          //hour 0 to 23
              minq = rand()%60,           //minute 0 to 59
              secq = rand()%60,           //second 0 to 59
              nanoq = rand()%1000000000,  //nanosecond 0 to 999999999
              tq = (((dayq*24+hourq)*60+minq)*60+secq)*1000000000+nanoq;
              result = ktj(-KN,tq);
              }break;
```

Minute

A minute atom is created using the *ka(-KU)* function. A minute is then specified using variables, as seen below. After which it's converted into seconds by preforming a simple calculation.

```
//minute
case 'u':{    int minq = rand()%60,       //minute 0 to 59
              secq = rand()%60,           //second 0 to 59
              tq = minq*60+secq;
              result = ka(-KU);
              result->i = tq;
              }break;
```

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.

```
//second
case 'v':{    int hourq = rand()%24,      //hour 0 to 23
              minq = rand()%60,           //minute 0 to 59
              secondq = rand()%60,        //second 0 to 59
              tq = (hourq*60+minq)*60+secq;
              result = ka(-KV);
              result->i = tq;
              }break;
```

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.

```
//time
case 't':{    int hourq = rand()%24,      //hour 0 to 23
              minq = rand()%60,           //minute 0 to 59
              secq = rand()%60,           //second 0 to 59
              miliq = rand()%1000,        //millisecond 0 to 999
              tq = ((hourq*60+minq)*60+secq)*1000+miliq;
              result = kt(tq);
              }break;
```

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.

Mixed

A general mixed list is created using the *ktn(0,n)* function and a selection of random atoms are inserted into the list.

```
//mixed
case '0':{    char typec[]="bghijefcspmdznuvt";
              result = ktn(0,lenq);
              for(i=0;i<lenq;i++){

                     kK(result)[i] = makeatom(kc((int)typec[rand()%(sizeof(typec)-1)]));

              };
              }break;
```

Boolean

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

```
//boolean
case 'B':{    result = ktn(KB,lenq);
              for(i=0;i<lenq;i++){

                     kG(result)[i]=makeatom(kc((int)'b'))->g;

              };
              }break;
```

Int

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

```
//int
case 'I':{    result = ktn(KI,lenq);
              for(i=0;i<lenq;i++){

                     kI(result)[i]=makeatom(kc((int)'i'))->i;

              };
              }break;
```

Symbol

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

```
//symbol
case 'S':{    result = ktn(KS,lenq);
              for(i=0;i<lenq;i++){

                     kS(result)[i]=makeatom(kc((int)'s'))->s;

              };
              }break;
```

Timestamp

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

```
//timestamp
case 'P':{    result = ktn(KP,lenq);
              for(i=0;i<lenq;i++){

                     kJ(result)[i]=makeatom(kc((int)'p'))->j;

              };
              }break;
```

Timespan

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

```
//timespan
case 'N':{    result = ktn(KN,lenq);
              for(i=0;i<lenq;i++){

                     kJ(result)[i]=makeatom(kc((int)'n'))->j;

              };
              }break;
```

6.3 Creating a dictionary

As discussed above, a dictionary has 2 important components. These are the keys and values.

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 const char head[]="bghijefcspmdznuvt0";
static const char ty[]="BGHIJEFCSPMDZNUVT0";

//Construct the key of dictionary
K key = ktn(KS,sizeof(head)-1);

for(i=0;i<sizeof(head)-1;i++){
       char buf;
       sprintf(&buf,"%s%c","t_",head[i]);
       char* buff=&buf;
       kS(key)[i]=ss(buff);
}
```

The values of a dictionary are stored in a list of data and the data is constructed from a number of lists of atoms.

```
//Construct a list for each key
K val = ktn(0,sizeof(ty)-1);

for(ii=0;ii<sizeof(ty)-1;ii++){

       kK(val)[ii] = makelist(kc(ty[ii]));
}
```

After preparing the key and the value objects, the dictionary is then created using the *xD* function.

```
//Build the dictionary
K result = xD(key,val);
```

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.

```
K result = xT(makedic(x));
```

6.5 Creating a nested table

A nested table is similar to a normal table. A major difference is that instead of holding atoms, a nested list holds other lists. An example would be a column with a string list. This data is "nested" since a string is a list of characters.

```
//Construct a nested list for each key
K val = ktn(0,sizeof(ty)-1);

for(ii=0;ii<sizeof(ty)-1;ii++){

	K nest = ktn(0,3);
	for(j=0;j<3;j++){
		kK(nest)[j]=makelist(kc(ty[ii]));
	}

	kK(val)[ii]=nest;
}
```

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