The C - K Interface

Introduction

This document defines the API for calling C functions from K and K functions from C. The C functions that make up the API are listed in the table below, together with the sections in which they are described. C functions called from K must manage K arguments and produce K results. C programs that call K functions must create K arguments and manage K results. The API functions for managing K data are listed in the first 5 rows of the table. The next-to-last row lists API functions for calling K functions and accessing K data from C. The mechanism for calling C functions from K is part of the K language and therefore does not appear in this table; see Linking to C Functions in K".

K <-> C Interface Functions

C Function	Section Reference
`gi`, `gf`, `gc`, `gs`, `gn`, `sp`	Creating K Atoms, Data Types
`gtn`, `gnk`, `gp`, `gpn`, `gsk`	Creating K Lists
`Ki`, `Kf`, `Kc`, `Ks`	Accessing and Modifying K Atoms
`KI`, `KF`, `KC`, `KS`, `KK`, `kap`	Accessing and Modifying K Lists
`dj`, `jd`	Date Conversion
`kerr`	Signalling a K Error
`sdf`, `scd`	Registering K Event Loop Callbacks
`ksk`, `sfn`	Calling K From C
`cd`, `ci`	Managing Reference Counts

Compilation

If the main program is a K program, then the C functions that will be called from a K program must be defined as entry points in an NT DLL (file extension `.dll`) or a Unix SO (file extension `.so`). Otherwise, C functions to be called from K can also be defined in the C main program. If there is a C main program then the K environment must always be ini-

tialized before any K objects are created or K statements executed; see “Calling K from C”.

Header and Lib files

These files can be found at www.kx.com/a/k/connect. They are `K20.lib`, `K20.h` and `K20x.h`. Include `K20x.h` in your C files and use `K20.lib` for linking. There are brie comments in `K20.h` and `K20x.h` summarizing the contents of this document.

The C Structure of the K Data Object

The internal format of K data objects is defined in `K20.h` by the recursive C-structure named `K`. The relevant members of the K structure, which are all of type `int`, are

- `c` – reference count of the object
- `t` – the data type of the object
- `n` – the number of data items when the object is a list or dictionary

The structure members are primarily for reference. However, there are occasions when you must manage the reference count (see “Managing Reference Counts”).

Data Types

The data types of K objects are represented by integer values, as follows.

- `6` – null atom
- `5` – dictionary
- `4` – symbol atom, i.e. `sp(`null-terminated character string`)`[1]
- `3` – character atom (unsigned)
- `2` – double atom
- `1` – integer atom
- `0` – general list whose items are other K objects
- `-1` – integer list

1. The API function `sp` internalizes its character string argument in K for optimized searches, but does not create a K object.

- `-2` – double list
- `-3` – character list
- `-4` – symbol list, i.e. each item is `sp(`null-terminated character string`)`[1]

Creating K Atoms

There are atomic constructors for each type of K atom. They are

- `gi` – generate an integer atom, as in `gi(3)` or `gi(i)` for `int i`;
- `gf` – generate a floating-point atom, as in `gf(3.5)` or `gf(a)` for `double a`;
- `gc` – generate a character atom, as in `gc('c')` or `gc(a)` for `unsigned char a`;
- `gs` – generate a symbol atom, as in `gs(sp("price"))` or `gs(sp(s))` for `char*s`;
- `gn` – generate the null atom, as in `gn()`.

Creating K Lists

There are several list constructors. The most general is `gtn(type,count)`. For example, `gtn(-1,5)` creates an integer list of length 5. Valid data types are 0, -1, -2, -3, -4. Valid counts are non-negative integers.

The constructor `gnk` creates a list from its arguments. It is useful for small lists, particularly for building argument lists to K functions called from C. The first argument to `gnk` is the number of arguments that follow, which can be from zero to eight. It is also the length of the result list. The remaining arguments become the items of the result, in order. For example,

```
gnk(5,gi(2),gf(3.4),gc('a'),gs(sp("abc")),gn());
```

is a list of 5 items; the first item is 2, the second is 3.4, and so on. An argument other than first can be any K object.

Creating Character Lists

A K character list can be created from a null-terminated string using the constructor `gp`, as in

```
gp("abcd");
```

The constructor `gpn` is used to select a specific number of characters from the front of a C character array or string, as in

```
gpn(cv,10);
```

The result is a K character list of length 10.

Creating Dictionaries

A K dictionary is a list of type 5 consisting of symbol-value-attributes triples. A dictionary can be created with the API function `gtn`; for example, `gtn(5,10)` is a dictionary with `10` entries.

The API function `gsk` is a useful tool for creating symbol-value-(empty attributes) triples. For example,

```
gsk("abc",gf(2.71));
```

creates an item with symbol `‘abc` and value 2.71.

A dictionary can be treated as a general list (type 0) in C. Its type only matters in K. In particular, the access function for general lists, KK, can be used to access and replace dictionary items (see “The Access Function for a General List”) and the API function `kap` can be used to append a symbol-value-attributes triple to a dictionary (se Appending to a List”).

Accessing and Modifying K Atoms

If `x` is an integer atom then `Ki(x)` is a C `int`.

If `x` is a floating-point atom then `Kf(x)` is a C `double`.

If `x` is a character atom then `Kc(x)` is a C `unsigned char`.

If `x` is a string atom (symbol) the `Ks(x)` is a C `char*`.

The value of an atom can also be modified, as in

```
Ki(x)=2;
```

and

```
Ks(x)=sp("abc");
```

Accessing and Modifying K Lists

If `x` is a K integer list then th `i`th item `KI(x)[i]` is a C `int`.

If `x` is a K floating-point list then the `i`th item `KF(x)[i]` is a C `double`.

If `x` is a K character list then the `i`th item `KC(x)[i]` is a C `unsigned cha` .

If `x` is a K string (symbol) list the `i`th item `KS(x)[i]` is a C `char*`.

A list item can also be modified, as in

```
KF(x)[2]=3.5;
```

The Access Function for a General List

The access function for a general list is denoted by `KK`. It applies to K objects of type 0. The `i`th item of a K object `x` of type 0, `KK(x)[i]`, is also a K object. If, for example, that item is an integer list, then the items of that item can be accessed by `KI(KK(x)[i])[j]`.

Appending to a K List

The API function for appending to a K list is `kap`. Use this function if the length of th result is not known when the list is created. Start with an empty list, as in

```
x=gtn(-1,0);
```

which is an empty integer list. Whenever a new item to be appended is available, append it to `x` with `kap`. For example, append the value of C `int a` to `x` as follows.

```
kap(&x,&a);
```

If `y` is a general list (type 0) then the second argument of `kap` can be a pointer to any other K object; that K object becomes a new item of `y`.

`Note that kap` is item-to-list append, not list-to-list.

Calling K from C

K can be started from a C program, K scripts can be then be loaded and K expressions can be executed with results returned to C. The API function that does all this is `ksk`.

First of all, K must be initialized with`ksk("",0)`. Note that this function always returns a result. If the result is meaningless, as it is in the K initialization statement, it can be immediately freed using the API function `cd`; see “Managing Reference Counts”. You will often see C statements, like the following, that call `ksk` and immediately free the result with `cd`.

```
cd(ksk("",0));
```

The character string argument can hold any valid K expression or command. For example,

```
r=ksk("2+3",0);
```

or

```
cd(ksk("\\l script.k",0));
```

The character string can contain a K function definition or the name of an existing K function, in which case the second argument must be K list of the function arguments. For example,

```
ksk("{x+y}",gnk(2,gi(2),gi(3)));
```

produces the K integer atom 5.

K Errors in ksk Evaluations

If an error occurs in an evaluation by `ksk`, a variant of the null value is returned with the error message in the structure member named `n`. (If the null value is returned and this member is 0, then the K expression executed correctly and returned the null). For example,

```
r=ksk(e,a);
if(6==r->t&&NULL!=r->n)printf("err: %s\n",r->n);
```

C calling K calling C

It is conceivable that an application has a C main program that calls K functions that, in turn, call C functions. In this case it is not necessary to compile a separate library that is linked into K. The functions to be called from K can be compiled with the main program and registered as callbacks with K from the main program.

For example, suppose the C function is

```
K f(K x,K y){gi(Ki(x)+Ki(y));}
```

`f` can be registered in K using the API function `sfn`, as follows.

```
sfn("g",f,2);
```

where `g` is the name by which `f` is called from K and 2 is the number of arguments of `f`. The K function can now be called with `ksk`.

```
ksk("g[2;3]",0);
```

or

```
ksk("g",gnk(2,gi(2),gi(3)));
```

Linking to C Functions in K

The primitive dyadic K function `2:` defines links to C functions. A result of `2:` is a K function that, when called, calls the C function to which it is linked. The left argument to `2:` names the DLL or SO file (with path) in which the C function is found. The right argument is a pair whose first item is a string holding the name of the C function and whose second argument is the number of arguments to the C function.

For example, the C function

```
K f(K x,K y){....}
```

is linked into K with

```
g:obj 2: ("f"; 2)
```

The K program that calls `f` is named `g`.

It is up to the programmer to make sure that the arguments to the K function correctly match the arguments of the C function.

Signalling a K Error in a C function

A K error can be signalled by a C function called from K with the API function`kerr`. For example,

```
if(0>=x->t)return kerr("x must be an atom");
```

The effect, which is to signal an error in K with the message “x must be an atom”, is the same as if the error was signalled by a K function. Note that it may be necessary to clean up work-in-progress just before an error is signalled; see “Managing Reference Counts”.

When `kerr` is called, K internally copies the text string to a K object for reporting the error. The function kerr returns 0. Consequently, if the C function that reports the error (say, `f`) is called by another C function, the calling function can test whether an error was reported in `f` or a K object was returned with `0==f(...)`.

Managing Reference Counts

Reference counting is a standard technique in data management to avoid unnecessary copies of data. A K object has reference count 1 when it is created. Each independent use o an object after the first use causes the reference count to be incremented. When the current use of the object is no longer needed its reference count is decremented. When the reference count becomes 0 the space occupied by the object can be reused.

An *independent use* of a K object is one that potentially leads to a reference count decrement in the future. For example, suppose the object `a` is created and inserted in the object `b`. This is the first use of `a` and its reference count, which is 1, accounts for this use. Suppose `a` is now inserted in the object `c`. This is an independent use of `a` because a future use of `c` can cause the reference count of `a` to be decremented, independent of what happens to `b`. Consequently, the reference count o `a` must be incremented. The reference count of `a` must be incremented even if `a` is inserted a second time into `b`, because a future use of `b` can now cause either instance of `a` to be replaced (and thereby have its reference count decremented), independent of the other instance.

K manages objects created in K, including arguments passed to C functions from K. As long as there are no independent uses of the arguments within the C program, nothing must be done with regard to their reference counts. However, when K objects are created in C programs, there are circumstances when you must manage the reference counts of those objects.

A K object created in C function called from K and returned as that function’s result will be managed by K from the point of return onwards. Nothing must be done with regard to the reference count of htis object as long as there are no other independent uses of the object. However, if temporary K objects that are not part of the result are constructed in the function, or if a K result is under construction when an error is signalled, then the ref

erence counts of those objects must be decremented before the function returns. Their reference counts then decrease by 1 to 0, indicating that the storage allocated to these objects can be reused.

Similarly, when a K function is called from a C program, its result is returned to the C program. The reference count of this result must be decremented when the result is no longer in use.

Reference counts are decremented by the API function `cd`; for example, `cd(x)` decrements the reference count of the K object `x`.

Knowing when to decrement reference counts is analogous to knowing when to free temporary storage allocated with `malloc()`, but trickier because `cd` is recursive. For example, every item in a general K list (type 0) is also a K object with its own reference count. If a K object `x` is created and then inserted in the general K list `y`, and if the reference count of `y` is subsequently decremented, the reference count of `x` will be decremented automatically (and therefore should not also be decremented explicitly).

Note that examples in the text do not necessarily account for reference counts. For instance, the expression `f(gnk(3,x,y,z))` shows a C function `f` with a K argument. While this expression may serve the purpose of an example, the following must be done in practice.

```
t=gnk(3,x,y,z);
f(t);
cd(t);
```

Reference counts are incremented by the API function `ci`. For convenience, the function `ci` returns its reference-count-incremented argument as its result. For example, suppose that the K object `r` is the result of a C function called from K, but `r` is not created in that function. This is an independent use of `r` and therefore the reference count of `r` must be incremented. Incrementing `r` can be conveniently done in the `return` statement, as follows.

```
return ci(r);
```

It is best to avoid complicated reference count situations and leave memory management to K by moving global K objects that are created in C to the K side of the interface and referencing them there. For example, the K object `bonddata` can be moved to a global variable with the same name in the `.u` directory of K by

```
t=gnk(1,bonddata); r=ksk("{.u.bonddata::x}",t); cd(t)
```

The K object `.u.bonddata` can now be used in any K function and, in particular, ones called from C. K now manages this object, so that there is not need to explicitly decrement and increment reference counts in C.

Date Conversion

Both `jd` and `dj` take a C `int` argument and return a C `int` result. The argument to `jd` is an integer of the form `yyyymmdd` and the result is a Julian day count. The argument to`dj` is a day count and the result is a `yyyymmdd` integer. These functions are useful for dat arithmetic. For example, to add 5 days to a date, first convert to Julian days with`jd`, add 5 to the result and convert back with `dj`, as in

```
d=dj(jd(d)+5);
```

Registering K Event Loop Callbacks

It is possible to send and receive non-K IPC messages in a K application by managing the non-K connection in C functions and registering the socket callbacks in the K event loop with the API function `sdf`. This function takes two arguments, the socket id (for an *accept* callback) and the callback function, as in

```
sdf(sockid,fn);
```

Use the negative of the socket id to establish a *read* callback, as in

```
sdf(-sockid,gn);
```

The callback functions `fn` and `gn` both take one argument, which is the socket id. Close the socket with the API function `scd`, e.g. `scd(sockid)`.

Examples

Summing Two K objects

The following C function illustrates straightforward manipulation of K objects by sum ming two K integer objects, both of which are either an atom or list. The function header is for a DLL. The best way to sum two K objects is, of course, using K, as in

```
a = gtn(2,x,y); s = ksk("+",a); cd(a);
```

but the following function is instructive.

```
__declspec(dllexport) K my_sum(K x,K y)
{  K z;
   int i;
   // case: both x and y are atoms
   if(1==x->t&&1==y->t) return gi(Ki(x)+Ki(y));
   // case: x is an atom and y is a list
   if(1==x->t&&-1==y->t){
      K z=gtn(-1,y->n);  // z is the same length as y
      for(i=0;i<y->n;i++)KI(z)[i]=Ki(x)+KI(y)[i];
      return z;
   }
   // other cases: list x, atom y and lists x, y
}
```

It is left to the reader to complete this example.

Accessing a Kdb Server with KDBC

The following example can be found in www.kx.com/a/kdb/connect/kdbc.txt. It illustrates communication between a C program and a Kdb database server. The Kdb server, listening on port 2001, is started with the following command, which also creates the database from the SQL script `sp.s` (in the Kdb download from the kx website).

```
k db sp.s -p 2001
```

The C program connects to the database server, sends it an SQL query and processes the result. Note that the K result returned by the Kdb server is a 3-item general list holding the column names of the result table, the data in column order and the data types.

```
#include "k20x.h"
extern printf(S s,...),gets(S);
main()
{  K q,r,n,d,t; // q=query, r=query result,
                // n=column names, d=data, t=data types

 cd(ksk("h:3:(`;2001)",0)); // connect to the server
 cd(ksk("k:{h 4:x}",0));// remote execution function
 q=gnk(1,gp("select sum qty by p from sp")); // KSQL query
 r=ksk("k",q),cd(q);        // query result, free query
 n=KK(r)[0],d=KK(r)[1];     // names, inverted data
 t=KK(r)[2];                // data types
 printf("columns: %d\n",n->n);// number of columns
 {I i=0;for(;i<n->n;++i)// column names and types
    printf("%s %s\n",KS(n)[i],KS(t)[i]);}
 printf("rows: %d\n",KK(d)[0]->n); // number of rows
 printf("%s %d\n",KS(KK(d)[0])[0],
    KI(KK(d)[1])[0]);           // first row(varchar,int)
 cd(r);                         // free the result
 cd(ksk("3:h",0));              // close the connection
 {C b[1];printf("\ndone ... ");gets(b);} // prompt
 return 0;}
```

Evaluating KSQL Statements

KSQL stored procedures written in C use the Kdb entry point **.d.r** to evaluate KSQL statements. For example, the following C character string holds a KSQL **update** statement.

```
char s[]="update qty:2*qty from 'OrderItems' \
     where orderid=6099"
```

This constant is placed in a K character string as follows.

```
K v=gp(s);
```

The function **.d.r** is then called as follows.

```
K a=gnk(0,v);
cd(ksk(".d.r",a)); cd(a);
```

The result of `ksk` is immediately freed with `cd` because this **update** statement does not produce a useful result. Freeing the K argument list `a` also frees the character list `v`.

A Remote Procedure Call

The KDBC message format for a bulk update is

(`insert;(`tablename;bulk_data))

(see *Bulk Updates* in the Kdb Programming Guid). In this case the remote procedure is **insert**. The corresponding K message has a slightly different arrangement. First, the arguments to the remote procedure, which in this example is **insert**, must be grouped. Also, **insert** requires that its table name be a symbol, and therefore *"table"* must be replaced with a symbol. Assuming the bulk data has already been constructed as the K object `bd`, the **inser** message can be constructed as follows.

```
msg = gnk(1,gnk(2,gs(sp("insert")),
          gnk(2,gs(sp("tablename")),bd)));
```

The message can be sent to the Kdb server using the K function **k** defined in the previous example.

```
cd(ksk("k",msg));
```

The reference count of the `ksk` result is decremented immediately because it will not be used. The reference count of the message should also be decremented.

```
cd(msg);
```