

Elektra  
0.8.1

Generated by Doxygen 1.7.1

Sat Jun 9 2012 14:24:25



# Contents



# Chapter 1

## The Elektra API

### 1.1 Overview

Elektra provides a universal and secure framework to store configuration parameters in a global, hierarchical key database. The core is a small library implemented in C. The plugin-based framework fulfills many configuration-related tasks to avoid any unnecessary code duplication across applications while it still allows the core to stay without any external dependency. Elektra abstracts from cross-platform-related issues with a consistent API, and allows applications to be aware of other applications' configurations, leveraging easy application integration.

See the website for more information <http://www.libelektra.org>

### 1.2 API docu

This document occupies with the API implementation, documentation, internals and plugins. On the one hand it gives an overview and an introduction for developers using Elektra, on the other hand it gives an informal descriptions what methods must and may provide to allow an alternative implementation of the API.

The latest version of this document can be found at <http://doc.libelektra.org/api/current/html>

### 1.3 Using the Elektra Library

A C or C++ source file that wants to use Elektra should include:

```
#include <kdb.h>
```

To link an executable with the Elektra library, the correct way is to use the `pkg-config` tool:

```
bash$ cc `pkg-config --libs elektra` -o myapp myapp.c
```

## 1.4 Elektra API

The API was written in pure C because Elektra was designed to be useful even for the most basic system programs, which are all made in C. Also, being C, bindings to other languages can appear easily.

The API follows an Object Oriented design, and there are 3 main classes as shown by the figure:

Some general things you can do with each class are:

### KDB

- [The four lowlevel functions](#)
- [Open](#) and [Close](#) the Database
- [Get](#) and [Set](#) [KeySet](#) in the Database
- See [class documentation](#) for more

### Key

- Get and Set key properties like [name](#) , [string](#) or [binary](#) values, [permissions](#) , [changed time](#) and [comment](#)
- Test if it is a [user/](#) or [system/](#) key, etc
- See [class documentation](#) for more

### KeySet

- Linked list of Key objects
- Append [a single key](#) or an entire [KeySet](#)
- [Work with](#) its [internal cursor](#)
- See [class documentation](#) for more

## 1.5 Key Names and Namespaces

There are 2 trees of keys: `system` and `user`

- The "system" Subtree It is provided to store system-wide configuration keys, that is, configurations that daemons and system services will use. But all other programs will also try to fetch system keys to have a fallback managed by the distributor or admin when the user does not have configuration for its own.
- The "user" Subtree Used to store user-specific configurations, like the personal settings of a user to certain programs. The user subtree will always be favoured if present (except for security concerns the user subtree may not be considered). See [Cascading](#) in the documentation of [ksLookupByName\(\)](#) how the selection of user and system keys works.

## 1.6 Rules for Key Names

When using Elektra to store your application's configuration and state, please keep in mind the following rules:

- You are not allowed to create keys right under `system` or `user`. They are reserved for more generic purposes.
- The keys for your application, called say *MyApp*, should be created under `system/sw/MyApp/current` and `user/sw/MyApp/current`
- `current` is the default configuration profile, users may symlink to the profile they want.
- That means you just need to `kdbGet()` `system/sw/MyApp/profile` and `user/sw/MyApp/profile` and then `ksLookupByName()` in `/sw/MyApp/profile` while profile defaults to `current`, but may be changed by the user or admin. See [Cascading](#) to learn more about that feature.

## 1.7 Backend Overview

The core of elektra does not store configuration itself to the harddisk. Instead this work is delegated to backends.

If you want to develop a backend, you should already have some experience with Elektra from the user point of view. You should be familiar with the data structures: [Key](#) and [KeySet](#). Then you can start reading about Backends, which are composed out of [Plugins :: Elektra framework for plugins](#).

## 1.8 Glossary

- **pop**, used in `ksPop()` and `KDB_O_POP` means to remove a key from a keyset.
- **delete**, or abbr. `del`, used in `keyDel()`, `ksDel()` and `KDB_O_DEL` means to free a key or keyset. The memory can be used for something else afterwards.
- **remove** means that the key/value information in the physical database will be removed permanently.





## Chapter 2

# Todo List

Global **elektraDocGet**(Plugin \*handle, KeySet \*returned, Key \*parentKey)  
needs some updates

Global **keyClearSync**(Key \*key) Should be done only in **kdbGet()** part of plugins.



## Chapter 3

# Deprecated List

Global **KDB\_O\_SORT** dont use

Global **keyGetATime**(const Key \*key) This API is obsolete.

Global **keyGetCTime**(const Key \*key) This API is obsolete.

Global **keyGetGID**(const Key \*key) This API is obsolete.

Global **keyGetMode**(const Key \*key) This API is obsolete.

Global **keyGetMTime**(const Key \*key) This API is obsolete.

Global **keyGetUID**(const Key \*key) This API is obsolete.

Global **keySetATime**(Key \*key, time\_t atime) This API is obsolete.

Global **keySetCTime**(Key \*key, time\_t ctime) This API is obsolete.

Global **keySetDir**(Key \*key) This API is obsolete.

Global **keySetGID**(Key \*key, gid\_t gid) This API is obsolete.

Global **keySetMode**(Key \*key, mode\_t mode) This API is obsolete. It is only a mapping to `keySetMeta(key, "mode", str)` which should be preferred.

Global **keySetMTime**(Key \*key, time\_t mtime) This API is obsolete.

Global **keySetUID**(Key \*key, uid\_t uid) This API is obsolete.



## Chapter 4

# Error

**Global `elektraDocSet`(Plugin \*handle, KeySet \*returned, Key \*parentKey)** In normal execution cases a positive value will be returned. But in some cases you are not able to set keys and have to return -1. If you declare `kdbcGetnoError()` you are done, but otherwise you have to set the cause of the error. (Will be added with 0.7.1)



## Chapter 5

# Module Index

### 5.1 Modules

Here is a list of all modules:

Elektra Modules :: Elektra framework for loading modules . . . . .	??
Interface for mounting backends . . . . .	??
Internal Datastructure for mountpoints . . . . .	??
KDB :: Low Level Methods . . . . .	??
KDB Backends :: Internal Helper for Elektra . . . . .	??
Key :: Basic Methods . . . . .	??
Key :: Meta Info Manipulation Methods . . . . .	??
Key :: Methods for Making Tests . . . . .	??
Key :: Name Manipulation Methods . . . . .	??
Key :: Value Manipulation Methods . . . . .	??
KeySet :: Class Methods . . . . .	??
Plugins :: Elektra framework for plugins . . . . .	??
Split :: Represents splitted keysets . . . . .	??





# Chapter 6

## Data Structure Index

### 6.1 Data Structures

Here are the data structures with brief descriptions:

<a href="#">_Backend</a>	. . . . .	??
<a href="#">_KDB</a>	. . . . .	??
<a href="#">_Key</a>	. . . . .	??
<a href="#">_KeySet</a>	. . . . .	??
<a href="#">_Plugin</a>	. . . . .	??
<a href="#">_Split</a>	. . . . .	??
<a href="#">_Trie</a>	. . . . .	??



# Chapter 7

## Module Documentation

### 7.1 KDB Backends :: Internal Helper for Elektra

Internal Methods for Elektra and Backends.

#### Functions

- int [elektraStrCmp](#) (const char \*s1, const char \*s2)
- int [elektraStrCaseCmp](#) (const char \*s1, const char \*s2)
- int [elektraRealloc](#) (void \*\*buffer, size\_t size)
- void [elektraFree](#) (void \*ptr)
- char \* [elektraStrDup](#) (const char \*s)
- char \* [elektraStrNDup](#) (const char \*s, size\_t l)
- size\_t [elektraStrLen](#) (const char \*s)

#### 7.1.1 Detailed Description

Internal Methods for Elektra and Backends. To use them:

```
#include <kdbbackend.h>
```

There are some areas where libraries have to reimplement some basic functions to archive support for non-standard systems, for testing purposes or to provide a little more convenience.

#### 7.1.2 Function Documentation

##### 7.1.2.1 void [elektraFree](#) ( void \* *ptr* )

Free memory of elektra or its backends.

**Parameters**

*ptr* the pointer to free

**See also**

elektraMalloc

**7.1.2.2 int elektraRealloc ( void \*\* *buffer*, size\_t *size* )**

Reallocate Storage in a save way.

```
if (elektraRealloc ((void **) & buffer, new_length) < 0) {
    // here comes the failure handler
    // you can still use the old buffer
#ifdef DEBUG
    fprintf (stderr, "Reallocation error\n");
#endif
    free (buffer);
    buffer = 0;
    // return with error
}
*
```

**Parameters**

*buffer* is a pointer to a malloc

*size* is the new size for the memory

**Returns**

-1 on failure

0 on success

**7.1.2.3 int elektraStrCaseCmp ( const char \* *s1*, const char \* *s2* )**

Compare Strings ignoring case using kdb semantics.

TODO: semantics not correct Does not work with binary sort.

**Parameters**

*s1* The first string to be compared

*s2* The second string to be compared

**Returns**

a negative number if *s1* is less than *s2*

0 if *s1* matches *s2*

a positive number if *s1* is greater than *s2*

#### 7.1.2.4 `int elektraStrCmp ( const char * s1, const char * s2 )`

Compare Strings using kdb semantics.

##### Parameters

*s1* The first string to be compared

*s2* The second string to be compared

/ is handled special, it will always be preferred for any other character.

##### Returns

a negative number if *s1* is less than *s2*

a number less than, equal to or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

#### 7.1.2.5 `char* elektraStrDup ( const char * s )`

Copy string into new allocated memory.

You need to free the memory yourself.

##### Note

that size is determined at runtime. So if you have a size information, don't use that function.

##### Parameters

*s* the null-terminated string to duplicate

##### Returns

0 if out of memory, a pointer otherwise

##### Precondition

*s* must be a c-string.

##### See also

[elektraFree](#)  
[elektraStrLen](#)  
[elektraStrNDup](#)

#### 7.1.2.6 `size_t elektraStrLen ( const char * s )`

Calculates the length in bytes of a string.

This function differs from `strlen()` because it is Unicode and multibyte chars safe. While `strlen()` counts characters and ignores the final NULL, [elektraStrLen\(\)](#) count bytes including the ending NULL.

**Parameters**

*s* the string to get the length from

**Returns**

number of bytes used by the string, including the final NULL.

**7.1.2.7 char\* elektraStrNDup ( const char \* *s*, size\_t *l* )**

Copy buffer into new allocated memory.

You need to free the memory yourself.

This function also works with \0 characters in the buffer. The length is taken as given, it must be correct.

**Returns**

0 if out of memory, a pointer otherwise

**Parameters**

*s* must be a allocated buffer

*l* the length of *s*

**7.2 KDB :: Low Level Methods**

General methods to access the Key database.

**Enumerations**

```
• enum option_t {
    KDB_O_NONE = 0, KDB_O_DEL = 1, KDB_O_POP = 1<<1, KDB_O_-
    NODIR = 1<<2,
    KDB_O_DIRONLY = 1<<3, KDB_O_NOREMOVE = 1<<6, KDB_O_-
    REMOVEONLY = 1<<7, KDB_O_INACTIVE = 1<<8,
    KDB_O_SYNC = 1<<9, KDB_O_SORT = 1<<10, KDB_O_-
    NORECURSIVE = 1<<11, KDB_O_NOCASE = 1<<12,
    KDB_O_WITHOWNER = 1<<13, KDB_O_NOALL = 1<<14 }
```

**Functions**

- KDB \* kdbOpen (Key \*errorKey)
- int kdbClose (KDB \*handle, Key \*errorKey)
- int kdbGet (KDB \*handle, KeySet \*ks, Key \*parentKey)
- int kdbSet (KDB \*handle, KeySet \*ks, Key \*parentKey)

### 7.2.1 Detailed Description

General methods to access the Key database. To use them:

```
#include <kdb.h>
```

The `kdb*()` class of methods are used to access the storage, to get and set [Keys](#) or [KeySets](#).

The most important functions are:

- [kdbOpen\(\)](#)
- [kdbClose\(\)](#)
- [kdbGet\(\)](#)
- [kdbSet\(\)](#)

The two essential functions for dynamic information about backends are:

- [kdbGetMountpoint\(\)](#)

They use some backend implementation to know the details about how to access the storage. Currently we have this backends:

- `berkeleydb`: the keys are stored in a Berkeley DB database, providing very small footprint, speed, and other advantages.
- `filesystem`: the key hierarchy and data are saved as plain text files in the filesystem.
- `ini`: the key hierarchy are saved into configuration files.

**See also**

<http://www.libelektra.org/Ini>

- `fstab`: a reference backend used to interpret the `/etc/fstab` file as a set of keys under `system/filesystems`.
- `gconf`: makes Elektra use the GConf daemon to access keys. Only the `user/` tree is available since GConf is not system wide.

Backends are physically a library named `/lib/libelektra-{NAME}.so`.

See [writing a new plugin](#) for information about how to write a plugin.

Language binding writers should follow the same rules:

- You must relay completely on the backend-dependent methods.
- You may use or reimplement the second set of methods.
- You should completely reimplement in your language the higher lever methods.
- Many methods are just for comfort in C. These methods are marked and need not to be implemented if the binding language has e.g. string operators which can do the operation easily.

## 7.2.2 Enumeration Type Documentation

### 7.2.2.1 enum option\_t

Options to change the default behavior of [kdbGet\(\)](#), [kdbSet\(\)](#) and [ksLookup\(\)](#) functions.

These options can be ORed. That is the |-Operator in C.

See also

[kdbGet\(\)](#), [kdbSet\(\)](#)

Enumerator:

***KDB\_O\_NONE*** No Option set. Will be recursive with no inactive keys.

See also

[kdbGet\(\)](#), [kdbSet\(\)](#), [ksLookup\(\)](#)

***KDB\_O\_DEL*** Delete parentKey key in [kdbGet\(\)](#), [kdbSet\(\)](#) or [ksLookup\(\)](#).

See also

[kdbGet\(\)](#), [kdbSet\(\)](#)

***KDB\_O\_POP*** Pop Parent out of keyset key in [kdbGet\(\)](#).

See also

[ksPop\(\)](#).

***KDB\_O\_NODIR*** Exclude keys containing other keys in result.

Only return leaves.

See also

[keyIsDir\(\)](#)

***KDB\_O\_DIRONLY*** Retrieve only directory keys (keys containing other keys).

This will give you an skeleton without leaves. This must not be used together with KDB\_O\_NODIR.

See also

[keyIsDir\(\)](#)

***KDB\_O\_NOREMOVE*** Don't remove any keys. This must not be used together with KDB\_O\_REMOVEONLY.

***KDB\_O\_REMOVEONLY*** Only remove keys. This must not be used together with KDB\_O\_NOREMOVE.

***KDB\_O\_INACTIVE*** Do not ignore inactive keys (that name begins with .).

See also

[keyIsInactive\(\)](#)

***KDB\_O\_SYNC*** Set keys independent of sync status.

See also

[keyNeedSync\(\)](#)



**KDB\_O\_SORT** This option has no effect. KeySets are always sorted.

**Deprecated**

dont use

**KDB\_O\_NORECURSIVE** Do not call [kdbGet\(\)](#) for every key containing other keys ([keyIsDir\(\)](#)).

**KDB\_O\_NOCASE** Ignore case.

**KDB\_O\_WITHOWNER** Search with owner.

**KDB\_O\_NOALL** Only search from start -> cursor to cursor -> end.

### 7.2.3 Function Documentation

#### 7.2.3.1 `int kdbClose ( KDB * handle, Key * errorKey )`

Closes the session with the Key database.

You should call this method when you finished your affairs with the key database. You can manipulate Key and KeySet objects also after [kdbClose\(\)](#). You must not use any kdb\* call afterwards. You can implement [kdbClose\(\)](#) in the `atexit()` handler.

This is the counterpart of [kdbOpen\(\)](#).

The `handle` parameter will be finalized and all resources associated to it will be freed. After a [kdbClose\(\)](#), this `handle` can't be used anymore, unless it gets initialized again with another call to [kdbOpen\(\)](#).

**See also**

[kdbOpen\(\)](#)

#### Parameters

***handle*** contains internal information of [opened](#) key database

***errorKey*** the key which holds error information

#### Returns

0 on success

-1 on NULL pointer

#### 7.2.3.2 `int kdbGet ( KDB * handle, KeySet * ks, Key * parentKey )`

Retrieve keys in an atomic and universal way, all other kdbGet Functions rely on that one.

The returned KeySet must be initialized. The returned KeySet may already contain some keys. The new retrieved keys will be appended using [ksAppendKey\(\)](#).

It will fully retrieve all keys under the `parentKey` folder, with all subfolders and their children.

### 7.2.4 Example

This example demonstrates the typical usecase within an application without updating.

**Example:**

```
KeySet *myConfig = ksNew(0);
Key *key = keyNew("system/sw/MyApp", KEY_END);
KDB *handle = kdbOpen(key);

kdbGet(handle, myConfig, key);

keySetName(key, "user/sw/MyApp");
kdbGet(handle, myConfig, key);

// check for errors in key
keyDel(key);

key = ksLookupByName(myConfig, "/sw/MyApp/key", 0);
// check if key is not 0 and work with it...

ksDel(myConfig); // delete the in-memory configuration

// maybe you want kdbSet() myConfig here

kdbClose(handle, 0); // no more affairs with the key database.
```

### 7.2.5 Details

When no backend could be found (e.g. no backend mounted) the default backend will be used.

If you pass NULL on any parameter [kdbGet\(\)](#) will fail immediately without doing anything.

When a backend fails [kdbGet\(\)](#) will return -1 without any changes to one of the parameter.

### 7.2.6 Updating

In the first run of [kdbGet](#) all keys are retrieved. On subsequent calls only the keys are retrieved where something was changed inside the key database. The other keys stay unchanged in the keyset, even when they were manipulated.

It is your responsibility to save the original keyset if you need it afterwards.

If you must get the same keyset again, e.g. in another thread you need to open a second handle to the key database using [kdbOpen\(\)](#).

**Parameters**

*handle* contains internal information of [opened](#) key database

*parentKey* parent key - invalid name gets all keys

*ks* the (pre-initialized) KeySet returned with all keys found will not be changed on error or if no update is required

#### See also

[ksLookupByName\(\)](#) for powerful lookups after the KeySet was retrieved

#### Returns

1 if the keys were retrieved successfully  
 0 if there was no update - no changes are made to the keyset then  
 -1 on failure - no changes are made to the keyset then

##### 7.2.6.1 KDB\* kdbOpen ( Key \* *errorKey* )

Opens the session with the Key database.

The first step is to open the default backend. With it system/elektra/mountpoints will be loaded and all needed libraries and mountpoints will be determined. These libraries for backends will be loaded and with it the KDB datastructure will be initialized.

You must always call this method before retrieving or committing any keys to the database. In the end of the program, after using the key database, you must not forget to [kdbClose\(\)](#). You can use the `atexit ()` handler for it.

The pointer to the KDB structure returned will be initialized like described above, and it must be passed along on any `kdb*()` method your application calls.

Get a KDB handle for every thread using `elektra`. Don't share the handle across threads, and also not the pointer accessing it:

```
thread1 {
    KDB * h;
    h = kdbOpen(0);
    // fetch keys and work with them
    kdbClose(h, 0);
}
thread2 {
    KDB * h;
    h = kdbOpen(0);
    // fetch keys and work with them
    kdbClose(h, 0);
}
```

You don't need to use the [kdbOpen\(\)](#) if you only want to manipulate plain in-memory Key or KeySet objects without any affairs with the backend key database or when your application loads plugins directly.

#### Parameters

*errorKey* the key which holds errors and warnings which were issued must be given

#### See also

[kdbClose\(\)](#) to end all affairs to the [Key :: Basic Methods](#) database.

**Returns**

a KDB pointer on success  
 NULL on failure

**7.2.6.2 int kdbSet ( KDB \* *handle*, KeySet \* *ks*, Key \* *parentKey* )**

Set keys in an atomic and universal way.

All other kdbSet Functions rely on that one.

**7.2.7 parentKey**

With parentKey you can only store a part of the given keyset.

```
KeySet *ks = ksNew(0);
Key *parentKey = keyNew("user/app/myapp/default", KEY_END);
kdbGet (h, ks, parentKey);

//now only set everything below user
if (kdbSet (h, ks, parentKey) == -1)
{
    // in parentKey you can check the error cause
    // ksCurrent(ks) is the faulty key
}

ksDel (ks);
```

If you pass a parentKey without a name the whole keyset will be set in an atomic way.

**7.2.8 Update**

Each key is checked with [keyNeedSync\(\)](#) before being actually committed. So only changed keys are updated. If no key of a backend needs to be synced any affairs to backends omitted and 0 is returned.

**7.2.9 Error Situations**

If some error occurs, [kdbSet\(\)](#) will stop. In this situation the KeySet internal cursor will be set on the key that generated the error.

None of the keys are actually committed.

You should present the error message to the user and let the user decide what to do. Possible solutions are:

- repeat the same kdbSet (for temporary errors)
- remove the key and set it again (for validation or type errors)
- change the value and try it again (for validation errors)

- do a kdbGet and then (for conflicts ...)
  - set the same keyset again (in favour of what was set by this user)
  - drop the old keyset (in favour of what was set elsewhere)
- export the configuration into a file (for unresolvable errors)

#### Example of how this method can be used:

```
int i;
KeySet *ks; // the KeySet I want to set
// fill ks with some keys
for (i=0; i< NR_OF_TRIES; i++) // limit to NR_OF_TRIES tries
{
    ret=kdbSet(handle, ks, parentKey);
    if (ret == -1)
    {
        // We got an error. Warn user.
        Key *problemKey = ksCurrent(ks);
        // parentKey has the errorInformation
        // problemKey is the faulty key (may be null)
        int userInput = showElektraErrorDialog (parentKey, problemKey);
        switch (userInput)
        {
            case INPUT_REPEAT: continue;
            case INPUT_REMOVE: ksLookup (ks, parentKey, KDB_O_POP); break;
            ...
        }
    }
}
```

#### Parameters

**handle** contains internal information of [opened](#) key database

**ks** a KeySet which should contain changed keys, otherwise nothing is done

**parentKey** holds the information below which key keys should be set, see above

#### Returns

1 on success  
 0 if nothing had to be done  
 -1 on failure

#### See also

[keyNeedSync\(\)](#), [ksNext\(\)](#), [ksCurrent\(\)](#)

## 7.3 Key :: Basic Methods

Key construction and initialization methods.

## Enumerations

- enum `keyswitch_t` {  
    `KEY_NAME` = 1, `KEY_VALUE` = 1<<1, `KEY_OWNER` = 1<<2, `KEY_COMMENT` = 1<<3,  
    `KEY_BINARY` = 1<<4, `KEY_UID` = 1<<5, `KEY_GID` = 1<<6, `KEY_MODE` = 1<<7,  
    `KEY_ETIME` = 1<<8, `KEY_MTIME` = 1<<9, `KEY_CTIME` = 1<<10,  
    `KEY_SIZE` = 1<<11,  
    `KEY_DIR` = 1<<14, `KEY_END` = 0 }

## Functions

- `Key * keyNew` (const char \*keyName,...)
- `Key * keyDup` (const `Key` \*source)
- `int keyCopy` (`Key` \*dest, const `Key` \*source)
- `int keyDel` (`Key` \*key)
- `ssize_t keyIncRef` (`Key` \*key)
- `ssize_t keyDecRef` (`Key` \*key)
- `ssize_t keyGetRef` (const `Key` \*key)

### 7.3.1 Detailed Description

Key construction and initialization methods. To use them:

```
#include <kdb.h>
```

A `Key` is the essential class that encapsulates key `name` , `value` and `metainfo` . `Key` properties are:

- `Key name`
- `Key value`
- `Key comment`
- `Key owner`
- `UID, GID and filesystem-like mode permissions`
- `Mode, change and modification times`

Described here the methods to allocate and free the key.

## 7.3.2 Enumeration Type Documentation

### 7.3.2.1 enum keyswitch\_t

Switches to denote the various Key attributes in methods throughout this library.

This enum switch provide a flag for every metadata in a key.

In case of [keyNew\(\)](#) they give Information what Parameter comes next.

See also

[keyNew\(\)](#)  
[ksToStream\(\)](#), [keyToStream\(\)](#)

Enumerator:

**KEY\_NAME** Flag for the key name  
**KEY\_VALUE** Flag for the key data  
**KEY\_OWNER** Flag for the key user domain  
**KEY\_COMMENT** Flag for the key comment  
**KEY\_BINARY** Flag if the key is binary  
**KEY\_UID** Flag for the key UID  
**KEY\_GID** Flag for the key GID  
**KEY\_MODE** Flag for the key permissions  
**KEY\_ATIME** Flag for the key access time  
**KEY\_MTIME** Flag for the key change time  
**KEY\_CTIME** Flag for the key status change time  
**KEY\_SIZE** Flag for maximum size to limit value  
**KEY\_DIR** Flag for the key directories  
**KEY\_END** Used as a parameter terminator to [keyNew\(\)](#)

## 7.3.3 Function Documentation

### 7.3.3.1 int keyCopy ( Key \* *dest*, const Key \* *source* )

Copy or Clear a key.

Most often you may prefer [keyDup\(\)](#) which allocates a new key and returns a duplication of another key.

But when you need to copy into an existing key, e.g. because it was passed by a pointer in a function you can do so:

```
void h (Key *k)
{
    // receive key c
    keyCopy (k, c);
    // the caller will see the changed key k
}
```

The reference counter will not change for the destination key. Affiliation to keysets are also not affected.

When you pass a NULL-pointer as source the data of dest will be cleaned completely and you get a fresh dest key.

```
void g (Key *k)
{
    keyCopy (k, 0);
    // k is now an empty and fresh key
}
```

The meta data will be duplicated for the destination key. So it will not take much additional space, even with lots of metadata.

If you want to copy all metadata, but keep the old value you can use [keyCopy\(\)](#) too.

```
void j (Key *k)
{
    size_t size = keyGetValueSize (k);
    char *value = malloc (size);
    int bstring = keyIsString (k);

    // receive key c
    memcpy (value, keyValue(k), size);
    keyCopy (k, c);
    if (bstring) keySetString (k, value);
    else keySetBinary (k, value, size);
    free (value);
    // the caller will see the changed key k
    // with the metadata from c
}
```

### Note

Next to the value itself we also need to remember if the value was string or binary. So in fact the meta data of the resulting key k in that example is not a complete duplicate, because the meta data "binary" may differ. Similar considerations might be necessary for the type of the key and so on, depending on the concrete situation.

### Parameters

**dest** the key which will be written to

**source** the key which should be copied or NULL to clean the destination key

### Returns

-1 on failure when a NULL pointer was passed for dest or a dynamic property could not be written.

0 when dest was cleaned

1 when source was successfully copied

### See also

[keyDup\(\)](#) to get a duplication of a [Key :: Basic Methods](#)



### 7.3.3.2 ssize\_t keyDecRef ( Key \* key )

Decrement the viability of a key object.

The references will be decremented for [ksPop\(\)](#) or successful calls of [ksLookup\(\)](#) with the option KDB\_O\_POP. It will also be decremented with an following [keyDel\(\)](#) in the case that an old key is replaced with another key with the same name.

The reference counter can't be decremented once it reached 0. In that situation nothing will happen and 0 will be returned.

#### Note

[keyDup\(\)](#) will reset the references for dupped key.

#### Returns

the value of the new reference counter  
-1 on null pointer  
0 when the key is ready to be freed

#### Parameters

*key* the key object to work with

#### See also

[keyGetRef\(\)](#), [keyDel\(\)](#), [keyIncRef\(\)](#)

### 7.3.3.3 int keyDel ( Key \* key )

A destructor for Key objects.

Every key created by [keyNew\(\)](#) must be deleted with [keyDel\(\)](#).

It is save to delete keys which are in a keyset, the number of references will be returned then.

It is save to delete a nullpointer, -1 will be returned then.

#### Parameters

*key* the key object to delete

#### See also

[keyNew\(\)](#), [keyInc\(\)](#), [keyGetRef\(\)](#)

#### Returns

the value of the reference counter if the key is within keyset(s)  
0 when the key was freed  
-1 on null pointers

#### 7.3.3.4 Key\* keyDup ( const Key \* source )

Return a duplicate of a key.

Memory will be allocated as needed for dynamic properties.

The new key will not be member of any KeySet and will start with a new reference counter at 0. A subsequent [keyDel\(\)](#) will delete the key.

```
int f (const Key * source)
{
    Key * dup = keyDup (source);
    // work with duplicate
    keyDel (dup);
    // everything related to dup is freed
    // and source is unchanged
}
```

Like for a new key after [keyNew\(\)](#) a subsequent [ksAppend\(\)](#) makes a KeySet to take care of the lifecycle of the key.

```
int g (const Key * source, KeySet * ks)
{
    Key * dup = keyDup (source);
    // work with duplicate
    ksAppendKey (ks, dup);
    // ksDel(ks) will also free the duplicate
    // source remains unchanged.
}
```

Duplication of keys should be preferred to [keyNew\(\)](#), because data like owner can be filled with a copy of the key instead of asking the environment. It can also be optimized in the checks, because the keyname is known to be valid.

#### Parameters

**source** has to be an initialised source Key

#### Returns

0 failure or on NULL pointer  
a fully copy of source on success

#### See also

[ksAppend\(\)](#), [keyDel\(\)](#)  
[keyClear\(\)](#), [keyNew\(\)](#)

#### 7.3.3.5 ssize\_t keyGetRef ( const Key \* key )

Return how many references the key has.

The references will be incremented on successful calls to [ksAppendKey\(\)](#) or [ksAppend\(\)](#).

**Note**

[keyDup\(\)](#) will reset the references for dapped key.

For your own applications you can use [keyIncRef\(\)](#) and [keyDecRef\(\)](#) for reference counting. Keys with zero references will be deleted when using [keyDel\(\)](#).

**Parameters**

*key* the key object to work with

**Returns**

the number of references  
-1 on null pointer

**See also**

[keyIncRef\(\)](#) and [keyDecRef\(\)](#)

**7.3.3.6 ssize\_t keyIncRef ( Key \* *key* )**

Increment the viability of a key object.

This function is intended for applications using their own reference counter for key objects. With it you can increment the reference and thus avoid destruction of the object in a subsequent [keyDel\(\)](#).

```
Key *k;  
keyInc (k);  
function_that_keyDec(k);  
// work with k  
keyDel (k); // now really free it
```

The reference counter can't be incremented once it reached SSIZE\_MAX. In that situation nothing will happen and SSIZE\_MAX will be returned.

**Note**

[keyDup\(\)](#) will reset the references for dapped key.

**Returns**

the value of the new reference counter  
-1 on null pointer  
SSIZE\_MAX when maximum exceeded

**Parameters**

*key* the key object to work with

**See also**

[keyGetRef\(\)](#), [keyDecRef\(\)](#), [keyDel\(\)](#)

### 7.3.3.7 Key\* keyNew ( const char \* keyName, ... )

A practical way to fully create a Key object in one step.

This function tries to mimic the C++ way for constructors.

To just get a key object, simple do:

```
Key *k = keyNew(0);
// work with it
keyDel (k);
```

If you want the key object to contain a name, value, comment and other meta info read on.

#### Note

When you already have a key with similar properties its easier and cheaper to [keyDup\(\)](#) the key.

Due to ABI compatibility, the Key structure is not defined in kdb.h, only declared. So you can only declare pointers to Keys in your program, and allocate and free memory for them with [keyNew\(\)](#) and [keyDel\(\)](#) respectively. See <http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html#AEN135>

You can call it in many different ways depending on the attribute tags you pass as parameters. Tags are represented as the [keyswitch\\_t](#) values, and tell [keyNew\(\)](#) which Key attribute comes next.

The simplest and minimum way to use it is with no tags, only a key name:

```
Key *nullKey,*emptyNamedKey;

// Create a key that has no name, is completely empty, but is initialized
nullKey=keyNew(0);
keyDel (nullKey);

// Is the same as above
nullKey=keyNew("", KEY_END);
keyDel (nullKey);

// Create and initialize a key with a name and nothing else
emptyNamedKey=keyNew("user/some/example",KEY_END);
keyDel (emptyNamedKey);
```

[keyNew\(\)](#) allocates memory for a key object and cleans everything up. After that, it processes the given argument list.

The Key attribute tags are the following:

- [keyswitch\\_t::KEY\\_TYPE](#)

Next parameter is a type of the value. Default assumed is [KEY\\_TYPE\\_UNDEFINED](#). Set this attribute so that a subsequent [KEY\\_VALUE](#) can toggle to [keySetString\(\)](#) or [keySetBinary\(\)](#) regarding to [keyIsString\(\)](#) or [keyIsBinary\(\)](#). If you don't use [KEY\\_TYPE](#) but a [KEY\\_VALUE](#) follows afterwards, [KEY\\_TYPE\\_STRING](#) will be used.

- `keyswitch_t::KEY_SIZE`

Define a maximum length of the value. This is especially useful for setting a binary key. So make sure you use that before you `KEY_VALUE` for binary keys.

- `keyswitch_t::KEY_VALUE`

Next parameter is a pointer to the value that will be set to the key. If no `keyswitch_t::KEY_TYPE` was used before, `keyswitch_t::KEY_TYPE_STRING` is assumed. If `KEY_TYPE` was previously passed with a `KEY_TYPE_BINARY`, you should have passed `KEY_SIZE` before! Otherwise it will be cut off with first `\0` in string!

- `keyswitch_t::KEY_UID`, `keyswitch_t::KEY_GID`

Next parameter is taken as the UID (`uid_t`) or GID (`gid_t`) that will be defined on the key. See [keySetUID\(\)](#) and [keySetGID\(\)](#).

- `keyswitch_t::KEY_MODE`

Next parameter is taken as mode permissions (`mode_t`) to the key. See [keySetMode\(\)](#).

- `keyswitch_t::KEY_DIR`

Define that the key is a directory rather than a ordinary key. This means its executable bits in its mode are set. This option allows the key to have subkeys. See [keySetDir\(\)](#).

- `keyswitch_t::KEY_OWNER`

Next parameter is the owner. See [keySetOwner\(\)](#).

- `keyswitch_t::KEY_COMMENT`

Next parameter is a comment. See [keySetComment\(\)](#).

- `keyswitch_t::KEY_END`

Must be the last parameter passed to [keyNew\(\)](#). It is always required, unless the `keyName` is 0.

#### Example:

```
KeySet *ks=ksNew(0);

ksAppendKey(ks,keyNew(0));           // an empty key

ksAppendKey(ks,keyNew("user/sw",      // the name of the key
KEY_END));                          // no more args

ksAppendKey(ks,keyNew("user/tmp/ex1",
KEY_VALUE,"some data",              // set a string value
KEY_END));                          // end of args

ksAppendKey(ks,keyNew("user/tmp/ex2",
KEY_VALUE,"some data",              // with a simple value
KEY_MODE,0777,                      // permissions
KEY_END));                          // end of args

ksAppendKey(ks,keyNew("user/tmp/ex4",
```

```

        KEY_TYPE, KEY_TYPE_BINARY,          // key type
        KEY_SIZE, 7,                        // assume binary length 7
        KEY_VALUE, "some data",             // value that will be truncated in 7 byte
    S
        KEY_COMMENT, "value is truncated",
        KEY_OWNER, "root",                  // owner (not uid) is root
        KEY_UID, 0,                         // root uid
        KEY_END));                          // end of args

ksAppendKey(ks, keyNew("user/tmp/ex5",
    KEY_TYPE,
        KEY_TYPE_DIR | KEY_TYPE_BINARY, // dir key with a binary value
    KEY_SIZE, 7,
    KEY_VALUE, "some data",             // value that will be truncated in 7 byte
    S
        KEY_COMMENT, "value is truncated",
        KEY_OWNER, "root",              // owner (not uid) is root
        KEY_UID, 0,                     // root uid
        KEY_END));                      // end of args

ksDel(ks);

```

The reference counter (see [keyGetRef\(\)](#)) will be initialized with 0, that means a subsequent call of [keyDel\(\)](#) will delete the key. If you append the key to a keyset the reference counter will be incremented by one (see [keyInc\(\)](#)) and the key can't be deleted by a [keyDel\(\)](#).

```

Key *k = keyNew(0); // ref counter 0
ksAppendKey(ks, k); // ref counter of key 1
ksDel(ks); // key will be deleted with keyset
*

```

If you increment only by one with [keyInc\(\)](#) the same as said above is valid:

```

Key *k = keyNew(0); // ref counter 0
keyIncRef(k); // ref counter of key 1
keyDel(k);    // has no effect
keyDecRef(k); // ref counter back to 0
keyDel(k);    // key is now deleted
*

```

If you add the key to more keySets:

```

Key *k = keyNew(0); // ref counter 0
ksAppendKey(ks1, k); // ref counter of key 1
ksAppendKey(ks2, k); // ref counter of key 2
ksDel(ks1); // ref counter of key 1
ksDel(ks2); // k is now deleted
*

```

or use [keyInc\(\)](#) more than once:

```

Key *k = keyNew(0); // ref counter 0
keyIncRef(k); // ref counter of key 1
keyDel(k);    // has no effect
keyIncRef(k); // ref counter of key 2

```

```
keyDel (k);    // has no effect
keyDecRef(k); // ref counter of key 1
keyDel (k);    // has no effect
keyDecRef(k); // ref counter is now 0
keyDel (k);    // k is now deleted
*
```

they key won't be deleted by a [keyDel\(\)](#) as long refcounter is not 0.

The key's sync bit will always be set for any call, except:

```
Key *k = keyNew(0);
// keyNeedSync() will be false
```

### Parameters

**keyName** a valid name to the key, or NULL to get a simple initialized, but really empty, object

### See also

[keyDel\(\)](#)

### Returns

a pointer to a new allocated and initialized Key object, or NULL if an invalid keyName was passed (see [keySetName\(\)](#)).

## 7.4 Key :: Meta Info Manipulation Methods

Methods to do various operations on Key metainfo.

### Functions

- int [keyRewindMeta](#) (Key \*key)
- const Key \* [keyNextMeta](#) (Key \*key)
- const Key \* [keyCurrentMeta](#) (const Key \*key)
- int [keyCopyMeta](#) (Key \*dest, const Key \*source, const char \*metaName)
- int [keyCopyAllMeta](#) (Key \*dest, const Key \*source)
- const Key \* [keyGetMeta](#) (const Key \*key, const char \*metaName)
- ssize\_t [keySetMeta](#) (Key \*key, const char \*metaName, const char \*newMetaString)
- uid\_t [keyGetUID](#) (const Key \*key)
- int [keySetUID](#) (Key \*key, uid\_t uid)
- gid\_t [keyGetGID](#) (const Key \*key)
- int [keySetGID](#) (Key \*key, gid\_t gid)
- int [keySetDir](#) (Key \*key)
- mode\_t [keyGetMode](#) (const Key \*key)
- int [keySetMode](#) (Key \*key, mode\_t mode)

- `time_t keyGetATime (const Key *key)`
- `int keySetATime (Key *key, time_t atime)`
- `time_t keyGetMTime (const Key *key)`
- `int keySetMTime (Key *key, time_t mtime)`
- `time_t keyGetCTime (const Key *key)`
- `int keySetCTime (Key *key, time_t ctime)`

### 7.4.1 Detailed Description

Methods to do various operations on Key metainfo. To use them:

```
#include <kdb.h>
```

Next to [Name \(key and owner\)](#) and [Value \(data and comment\)](#) there is the so called meta information inside every key.

Key meta information are an unlimited number of key/value pairs strongly related to a key. Its main purpose is to give keys special semantics, so that plugins can treat them differently.

File system information (see `stat(2)` for more information):

- `uid`: the user id (positive number)
- `gid`: the group id (positive number)
- `mode`: filesystem-like mode permissions (positive octal number)
- `atime`: When was the key accessed the last time.
- `mtime`: When was the key modified the last time.
- `ctime`: When the uid, gid or mode of a key changes. (times are represented through a positive number as unix timestamp)

The comment can contain userdata which directly belong to that key. The name of the meta information is "comment" for a general purpose comment about the key. Multi-Language comments are also supported by appending [LANG] to the name.

Validators are regular expressions which are tested against the key value. The metakey "validator" can hold a regular expression which will be matched against.

Types can be expressed with the meta information "type".

The relevance of the key can be tagged with a value from -20 to 20. Negative numbers are the more important and must be present in order to start the program.

A version of a key may be stored with "version". Its format is full.major.minor where all of these are integers.

The order inside a persistent storage can be described with the tag "order" which contains a positive number.

The meta key "app" describes to which application a key belongs. It can be used to remove keys from an application no longer installed.



The meta key "path" describes where the key is physically stored.

The "owner" is the user that owns the key. It only works for the user/ hierarchy. It rather says where the key is stored and says nothing about the filesystem properties.

## 7.4.2 Function Documentation

### 7.4.2.1 `int keyCopyAllMeta ( Key * dest, const Key * source )`

Do a shallow copy of all meta data from source to dest.

The key dest will additionally have all meta data source had. Meta data not present in source will not be changed. Meta data which was present in source and dest will be overwritten.

For example the meta data type is copied into the Key k.

```
void l(Key *k)
{
    // receive c
    keyCopyMeta(k, c);
    // the caller will see the changed key k
    // with all the metadata from c
}
```

The main purpose of this function is for plugins or applications which want to add the same meta data to n keys. When you do that with [keySetMeta\(\)](#) it will take n times the memory for the key. This can be considerable amount of memory for many keys with some meta data for each.

To avoid that problem you can use [keyCopyAllMeta\(\)](#) or [keyCopyMeta\(\)](#).

```
void o(KeySet *ks)
{
    Key *current;
    Key *shared = keyNew (0);
    keySetMeta(shared, "shared1", "this meta data should be shared among many
keys");
    keySetMeta(shared, "shared2", "this meta data should be shared among many
keys also");
    keySetMeta(shared, "shared3", "this meta data should be shared among many
keys too");

    ksRewind(ks);
    while ((current = ksNext(ks)) != 0)
    {
        if (needs_shared_data(current)) keyCopyAllMeta(current, shared);
    }
}
```

### Postcondition

for every metaName present in source: `keyGetMeta(source, metaName) == keyGetMeta(dest, metaName)`

**Returns**

1 if was successfully copied  
 0 if source did not have any meta data  
 -1 on null pointers (source or dest)  
 -1 on memory problems

**Parameters**

*dest* the destination where the meta data should be copied too  
*source* the key where the meta data should be copied from

#### 7.4.2.2 **int keyCopyMeta ( Key \* *dest*, const Key \* *source*, const char \* *metaName* )**

Do a shallow copy of meta data from source to dest.

The key dest will have the same meta data referred with metaName afterwards then source.

For example the meta data type is copied into the Key k.

```
void l(Key *k)
{
    // receive c
    keyCopyMeta(k, c, "type");
    // the caller will see the changed key k
    // with the metadata "type" from c
}
```

The main purpose of this function is for plugins or applications which want to add the same meta data to n keys. When you do that with [keySetMeta\(\)](#) it will take n times the memory for the key. This can be considerable amount of memory for many keys with some meta data for each.

To avoid that problem you can use [keyCopyAllMeta\(\)](#) or [keyCopyMeta\(\)](#).

```
void o(KeySet *ks)
{
    Key *current;
    Key *shared = keyNew (0);
    keySetMeta(shared, "shared", "this meta data should be shared among many
keys");

    ksRewind(ks);
    while ((current = ksNext(ks)) != 0)
    {
        if (needs_shared_data(current)) keyCopyMeta(current, shared, "sha
red");
    }
}
```

**Postcondition**

keyGetMeta(source, metaName) == keyGetMeta(dest, metaName)

**Returns**

1 if was successfully copied  
0 if the meta data in dest was removed too  
-1 on null pointers (source or dest)  
-1 on memory problems

**Parameters**

*dest* the destination where the meta data should be copied too  
*source* the key where the meta data should be copied from  
*metaName* the name of the meta data which should be copied

**7.4.2.3 const Key\* keyCurrentMeta ( const Key \* key )**

Returns the Value of a Meta-Information which is current.

The pointer is NULL if you reached the end or after [ksRewind\(\)](#).

**Note**

You must not delete or change the returned key, use [keySetMeta\(\)](#) if you want to delete or change it.

**Parameters**

*key* the key object to work with

**Returns**

a buffer to the value pointed by *key*'s cursor  
0 on NULL pointer

**See also**

[keyNextMeta\(\)](#), [keyRewindMeta\(\)](#)  
[ksCurrent\(\)](#) for pedant in iterator interface of KeySet

**7.4.2.4 time\_t keyGetATime ( const Key \* key )**

Get last time the key data was read from disk.

**Deprecated**

This API is obsolete.

Every [kdbGet\(\)](#) might update the access time of a key. You get information when the key was read the last time from the database.

You will get 0 when the key was not read already.

Beware that multiple copies of keys with [keyDup\(\)](#) might have different atimes because you [kdbGet\(\)](#) one, but not the other. You can use this information to decide which key is the latest.

#### Parameters

*key* Key to get information from.

#### Returns

the time you got the key with [kdbGet\(\)](#)  
0 on key that was never [kdbGet\(\)](#)  
(time\_t)-1 on NULL pointer

#### See also

[keySetATime\(\)](#)  
[kdbGet\(\)](#)

#### 7.4.2.5 time\_t keyGetCTime ( const Key \* *key* )

Get last time the key metadata was changed from disk.

#### Deprecated

This API is obsolete.

You will get 0 when the key was not read already.

Any changed field in metadata will influence the ctime of a key.

This time is not updated if only value or comment are changed.

Not changed keys will not update this time, even after [kdbSet\(\)](#).

It is possible that other keys written to disc influence this time if the backend is not grained enough.

#### Parameters

*key* Key to get information from.

#### See also

[keySetCTime\(\)](#)

#### Returns

(time\_t)-1 on NULL pointer  
the metadata change time

#### 7.4.2.6 `gid_t keyGetGID ( const Key * key )`

Get the group ID of a key.

##### Deprecated

This API is obsolete.

#### 7.4.3 GID

The group ID is a unique identification for every group present on a system. Keys will belong to root (0) as long as you did not get their real GID with [kdbGet\(\)](#).

Unlike UID users might change their group. This makes it possible to share configuration between some users.

A fresh key will have (gid\_t)-1 also known as the group nogroup. It means that the key is not related to a group ID at the moment.

##### Parameters

*key* the key object to work with

##### Returns

the system's GID of the key  
(gid\_t)-1 on NULL key or currently unknown ID

##### See also

[keySetGID\(\)](#), [keyGetUID\(\)](#)

##### 7.4.3.1 `const Key* keyGetMeta ( const Key * key, const char * metaName )`

Returns the Value of a Meta-Information given by name.

This is a much more efficient version of [keyGetMeta\(\)](#). But unlike with [keyGetMeta](#) you are not allowed to modify the resulting string.

```
int f(Key *k)
{
    if (!strcmp(keyValue(keyGetMeta(k, "type")), "boolean"))
    {
        // the type of the key is boolean
    }
}
```

##### Note

You must not delete or change the returned key, use [keySetMeta\(\)](#) if you want to delete or change it.

**Parameters**

*key* the key object to work with  
*metaName* the name of the meta information you want the value from

**Returns**

0 if the key or metaName is 0  
0 if no such metaName is found  
value of Meta-Information if Meta-Information is found

**See also**

keyGetMetaSize(), [keyGetMeta\(\)](#), [keySetMeta\(\)](#)

**7.4.3.2 mode\_t keyGetMode ( const Key \* key )**

Return the key mode permissions.

**Deprecated**

This API is obsolete.

Default is 0664 (octal) for keys and 0775 for directory keys which used [keySetDir\(\)](#).  
The defaults are defined with the macros KEY\_DEF\_MODE and KEY\_DEF\_DIR.  
For more information about the mode permissions see [Modes](#).

**Parameters**

*key* the key object to work with

**Returns**

mode permissions of the key  
KEY\_DEF\_MODE as defaults  
(mode\_t)-1 on NULL pointer

**See also**

[keySetMode\(\)](#)

**7.4.3.3 time\_t keyGetMTime ( const Key \* key )**

Get last modification time of the key on disk.

**Deprecated**

This API is obsolete.

You will get 0 when the key was not read already.

Everytime you change value or comment and `kdbSet()` the key the mtime will be updated. When you `kdbGet()` the key, the atime is set appropriate.

Not changed keys may not even passed to `kdbSet_backend()` so it will not update this time, even after `kdbSet()`.

It is possible that other keys written to disc influence this time if the backend is not grained enough.

If you add or remove a key the key thereunder in the hierarchy will update the mtime if written with `kdbSet()` to disc.

#### Parameters

*key* Key to get information from.

#### See also

`keySetMTime()`

#### Returns

the last modification time  
(time\_t)-1 on NULL pointer

#### 7.4.3.4 uid\_t keyGetUID ( const Key \* key )

Get the user ID of a key.

#### Deprecated

This API is obsolete.

### 7.4.4 UID

The user ID is a unique identification for every user present on a system. Keys will belong to root (0) as long as you did not get their real UID with `kdbGet()`.

Although usually the same, the UID of a key is not related to its owner.

A fresh key will have no UID.

#### Parameters

*key* the key object to work with

#### Returns

the system's UID of the key  
(uid\_t)-1 on NULL key

**See also**

[keyGetGID\(\)](#), [keySetUID\(\)](#), [keyGetOwner\(\)](#)

**7.4.4.1 const Key\* keyNextMeta ( Key \* *key* )**

Iterate to the next meta information.

Keys have an internal cursor that can be reset with [keyRewindMeta\(\)](#). Every time [keyNextMeta\(\)](#) is called the cursor is incremented and the new current Name of Meta Information is returned.

You'll get a NULL pointer if the meta information after the end of the Key was reached. On subsequent calls of [keyNextMeta\(\)](#) it will still return the NULL pointer.

The *key* internal cursor will be changed, so it is not const.

**Note**

That the resulting key is guaranteed to have a value, because meta information has no binary or null pointer semantics.

You must not delete or change the returned key, use [keySetMeta\(\)](#) if you want to delete or change it.

**Parameters**

*key* the key object to work with

**Returns**

a key representing meta information  
0 when the end is reached  
0 on NULL pointer

**See also**

[ksNext\(\)](#) for pedant in iterator interface of KeySet

**7.4.4.2 int keyRewindMeta ( Key \* *key* )**

Rewind the internal iterator to first meta data.

Use it to set the cursor to the beginning of the Key Meta Infos. [keyCurrentMeta\(\)](#) will then always return NULL afterwards. So you want to [keyNextMeta\(\)](#) first.

```
Key *key;
const Key *meta;

keyRewindMeta (key);
while ((meta = keyNextMeta (key)) != 0)
{
    printf ("name: %s, value: %s", keyName(meta), (const char*)keyValue(meta)
    );
}
```



**Parameters**

*key* the key object to work with

**Returns**

0 on success

0 if there is no meta information for that key ([keyNextMeta\(\)](#) will always return 0 in that case)

-1 on NULL pointer

**See also**

[keyNextMeta\(\)](#), [keyCurrentMeta\(\)](#)

[ksRewind\(\)](#) for pedant in iterator interface of KeySet

**7.4.4.3 int keySetATime ( Key \* *key*, time\_t *atime* )**

Update the atime information for a key.

**Deprecated**

This API is obsolete.

When you do manual sync of keys you might also update the atime to make them indistinguishable.

It can also be useful if you work with keys not using a keydatabase.

**Parameters**

*key* The Key object to work with

*atime* The new access time for the key

**Returns**

0 on success

-1 on NULL pointer

**See also**

[keyGetATime\(\)](#)

**7.4.4.4 int keySetCTime ( Key \* *key*, time\_t *ctime* )**

Update the ctime information for a key.

**Deprecated**

This API is obsolete.

**Parameters**

- key* The Key object to work with  
*ctime* The new change metadata time for the key

**Returns**

- 0 on success  
-1 on NULL pointer

**See also**

[keyGetCTime\(\)](#)

**7.4.4.5 int keySetDir ( Key \* *key* )**

Set mode so that key will be recognized as directory.

**Deprecated**

This API is obsolete.

The function will add all executable bits.

- Mode 0200 will be translated to 0311
- Mode 0400 will be translated to 0711
- Mode 0664 will be translated to 0775

The macro KEY\_DEF\_DIR (defined to 0111) will be used for that.

The executable bits show that child keys are allowed and listable. There is no way to have child keys which are not listable for anyone, but it is possible to restrict listing the keys to the owner only.

- Mode 0000 means that it is a key not read or writable to anyone.
- Mode 0111 means that it is a directory not read or writable to anyone. But it is recognized as directory to anyone.

For more about mode see [keySetMode\(\)](#).

It is not possible to access keys below a not executable key. If a key is not writeable and executable [kdbSet\(\)](#) will fail to access the keys below. If a key is not readable and executable [kdbGet\(\)](#) will fail to access the keys below.

**Parameters**

- key* the key to set permissions to be recognized as directory.

**Returns**

0 on success  
-1 on NULL pointer

**See also**

[keySetMode\(\)](#)

**7.4.4.6 int keySetGID ( Key \* *key*, gid\_t *gid* )**

Set the group ID of a key.

**Deprecated**

This API is obsolete.

See [GID](#) for more information about group IDs.

**Parameters**

*key* the key object to work with  
*gid* is the group ID

**Returns**

0 on success  
-1 on NULL key

**See also**

[keyGetGID\(\)](#), [keySetUID\(\)](#)

**7.4.4.7 ssize\_t keySetMeta ( Key \* *key*, const char \* *metaName*, const char \* *newMetaString* )**

Set a new Meta-Information.

Will set a new Meta-Information pair consisting of *metaName* and *newMetaString*.

Will add a new Pair for Meta-Information if *metaName* was not added up to now.

It will modify a existing Pair of Meta-Information if the the *metaName* was inserted already.

It will remove a meta information if *newMetaString* is 0.

**Parameters**

*key* the key object to work with  
*metaName* the name of the meta information where you want to change the value

*newMetaString* the new value for the meta information

### Returns

-1 on error if key or metaName is 0, out of memory or names are not valid  
0 if the Meta-Information for metaName was removed  
size (>0) of newMetaString if Meta-Information was successfully added

### See also

[keyGetMeta\(\)](#)

#### 7.4.4.8 int keySetMode ( Key \* key, mode\_t mode )

Set the key mode permissions.

### Deprecated

This API is obsolete. It is only a mapping to [keySetMeta\(key, "mode", str\)](#) which should be preferred.

The mode consists of 9 individual bits for mode permissions. In the following explanation the octal notation with leading zero will be used.

Default is 0664 (octal) for keys and 0775 for directory keys which used [keySetDir\(\)](#).

The defaults are defined with the macros `KEY_DEF_MODE` and `KEY_DEF_DIR`.

### Note

libelektra 0.7.0 only allows 0775 (directory keys) and 0664 (other keys). More will be added later in a sense of the description below.

## 7.4.5 Modes

0000 is the most restrictive mode. No user might read, write or execute the key.

Reading the key means to get the value by [kdbGet\(\)](#).

Writing the key means to set the value by [kdbSet\(\)](#).

Execute the key means to make a step deeper in the hierarchy. But you must be able to read the key to be able to list the keys below. See also [keySetDir\(\)](#) in that context. But you must be able to write the key to be able to add or remove keys below.

0777 is the most relaxing mode. Every user is allowed to read, write and execute the key, if he is allowed to execute and read all keys below.

0700 allows every action for the current user, identified by the uid. See [keyGetUID\(\)](#) and [keySetUID\(\)](#).

To be more specific for the user the single bits can elect the mode for read, write and execute. 0100 only allows executing which gives the information that it is a directory

for that user, but not accessible. 0200 only allows reading. This information may be combined to 0300, which allows execute and reading of the directory. Last 0400 decides about the writing permissions.

The same as above is also valid for the 2 other octal digits. 0070 decides about the group permissions, in that case full access. Groups are identified by the gid. See [keyGetGID\(\)](#) and [keySetGID\(\)](#). In that example everyone with a different uid, but the gid of the the key, has full access.

0007 decides about the world permissions. This is taken into account when neither the uid nor the gid matches. So that example would allow everyone with a different uid and gid of that key gains full access.

#### Parameters

*key* the key to set mode permissions

*mode* the mode permissions

#### Returns

0 on success

-1 on NULL key

#### See also

[keyGetMode\(\)](#)

#### 7.4.5.1 int keySetMTime ( Key \* *key*, time\_t *mtime* )

Update the mtime information for a key.

#### Deprecated

This API is obsolete.

#### Parameters

*key* The Key object to work with

*mtime* The new modification time for the key

#### Returns

0 on success

#### See also

[keyGetMTime\(\)](#)

#### 7.4.5.2 int keySetUID ( Key \* key, uid\_t uid )

Set the user ID of a key.

#### Deprecated

This API is obsolete.

See [UID](#) for more information about user IDs.

#### Parameters

*key* the key object to work with

*uid* the user ID to set

#### Returns

0 on success

-1 on NULL key or conversion error

#### See also

[keySetGID\(\)](#), [keyGetUID\(\)](#), [keyGetOwner\(\)](#)

## 7.5 Key :: Name Manipulation Methods

Methods to do various operations on Key names.

### Functions

- const char \* [keyName](#) (const Key \*key)
- ssize\_t [keyGetNameSize](#) (const Key \*key)
- ssize\_t [keyGetName](#) (const Key \*key, char \*returnedName, size\_t maxSize)
- ssize\_t [keySetName](#) (Key \*key, const char \*newName)
- ssize\_t [keyGetFullNameSize](#) (const Key \*key)
- ssize\_t [keyGetFullName](#) (const Key \*key, char \*returnedName, size\_t maxSize)
- const char \* [keyBaseName](#) (const Key \*key)
- ssize\_t [keyGetBaseNameSize](#) (const Key \*key)
- ssize\_t [keyGetBaseName](#) (const Key \*key, char \*returned, size\_t maxSize)
- ssize\_t [keyAddBaseName](#) (Key \*key, const char \*baseName)
- ssize\_t [keySetBaseName](#) (Key \*key, const char \*baseName)
- const char \* [keyOwner](#) (const Key \*key)
- ssize\_t [keyGetOwnerSize](#) (const Key \*key)
- ssize\_t [keyGetOwner](#) (const Key \*key, char \*returnedOwner, size\_t maxSize)
- ssize\_t [keySetOwner](#) (Key \*key, const char \*newOwner)

### 7.5.1 Detailed Description

Methods to do various operations on Key names. To use them:

```
#include <kdb.h>
```

These functions make it easier for c programmers to work with key names. Everything here can also be done with `keySetName`, described in `key`.

#### Rules for Key Names

When using Elektra to store your application's configuration and state, please keep in mind the following rules:

- You are not allowed to create keys right under `system` or `user`.
- You are not allowed to create folder keys right under `system` or `user`. They are reserved for very essential OS subsystems.
- The keys for your application, called say *MyApp*, should be created under `system/sw/MyApp` and/or `user/sw/MyApp`.
- It is suggested to make your application look for default keys under `system/sw/MyApp/current` and/or `user/sw/MyApp/current`. This way, from a sysadmin perspective, it will be possible to copy the `system/sw/MyApp/current` tree to something like `system/sw/MyApp/old`, and keep system clean and organized.
- `\0` must not occur in names.
- `/` is the separator.

### 7.5.2 Function Documentation

#### 7.5.2.1 `ssize_t keyAddBaseName ( Key * key, const char * baseName )`

Adds `baseName` to the current key name.

Assumes that `key` is a directory. `baseName` is appended to it. The function adds `'/'` if needed while concatenating.

So if `key` has name `"system/dir1/dir2"` and this method is called with `baseName "mykey"`, the resulting key will have name `"system/dir1/dir2/mykey"`.

When `baseName` is 0 or "" nothing will happen and the size of the name is returned.

#### Warning

You should not change a keys name once it belongs to a keyset. See `ksSort()` for more information.

TODO: does not recognise .. and . in the string!

### Parameters

*key* the key object to work with  
*baseName* the string to append to the name

### Returns

the size in bytes of the new key name including the ending NULL  
-1 if the key had no name  
-1 on NULL pointers

### See also

[keySetBaseName\(\)](#)  
[keySetName\(\)](#) to set a new name.

#### 7.5.2.2 `const char* keyBaseName ( const Key * key )`

Returns a pointer to the real internal key name where the `basename` starts.

This is a much more efficient version of [keyGetBaseName\(\)](#) and you should use it if you are responsible enough to not mess up things. The name might change or even point to a wrong place after a [keySetName\(\)](#). If you need a copy of the `basename` consider to use [keyGetBaseName\(\)](#).

[keyBaseName\(\)](#) returns "" when there is no `keyBaseName`. The reason is

```
key=keyNew(0);  
keySetName(key, "");  
keyName(key); // you would expect "" here  
keySetName(key, "user");  
keyName(key); // you would expect "" here  
keyDel(key);
```

### Note

Note that the `Key` structure keeps its own `size` field that is calculated by library internal calls, so to avoid inconsistencies, you must never use the pointer returned by [keyBaseName\(\)](#) method to set a new value. Use [keySetBaseName\(\)](#) instead.

### Parameters

*key* the object to obtain the `basename` from

### Returns

a pointer to the `basename`  
"" when the key has no (base)name  
0 on NULL pointer



**See also**

[keyGetBaseName\(\)](#), [keyGetBaseNameSize\(\)](#)  
[keyName\(\)](#) to get a pointer to the name  
[keyOwner\(\)](#) to get a pointer to the owner

**7.5.2.3 `ssize_t keyGetBaseName ( const Key * key, char * returned, size_t maxSize )`**

Calculate the basename of a key name and put it in `returned` finalizing the string with NULL.

Some examples:

- `basename of system/some/keyname` is `keyname`
- `basename of "user/tmp/some key"` is `"some key"`

**Parameters**

***key*** the key to extract basename from  
***returned*** a pre-allocated buffer to store the basename  
***maxSize*** size of the `returned` buffer

**Returns**

number of bytes copied to `returned`  
1 on empty name  
-1 on NULL pointers  
-1 when `maxSize` is 0 or larger than `SSIZE_MAX`

**See also**

[keyBaseName\(\)](#), [keyGetBaseNameSize\(\)](#)  
[keyName\(\)](#), [keyGetName\(\)](#), [keySetName\(\)](#)

**7.5.2.4 `ssize_t keyGetBaseNameSize ( const Key * key )`**

Calculates number of bytes needed to store basename of `key`.

Key names that have only root names (e.g. `"system"` or `"user"` or `"user:domain"`) does not have basenames, thus the function will return 1 bytes to store `""`.

Basenames are denoted as:

- `system/some/thing/basename -> basename`
- `user:domain/some/thing/base\name > base\name`

**Parameters**

*key* the key object to work with

**Returns**

size in bytes of *key*'s basename including ending NULL

**See also**

[keyBaseName\(\)](#), [keyGetBaseName\(\)](#)  
[keyName\(\)](#), [keyGetName\(\)](#), [keySetName\(\)](#)

**7.5.2.5 `ssize_t keyGetFullName ( const Key * key, char * returnedName, size_t maxSize )`**

Get key full name, including the user domain name.

**Returns**

number of bytes written  
1 on empty name  
-1 on NULL pointers  
-1 if maxSize is 0 or larger than SSIZE\_MAX

**Parameters**

*key* the key object  
*returnedName* pre-allocated memory to write the key name  
*maxSize* maximum number of bytes that will fit in returnedName, including the final NULL

**7.5.2.6 `ssize_t keyGetFullNameSize ( const Key * key )`**

Bytes needed to store the key name including user domain and ending NULL.

**Parameters**

*key* the key object to work with

**Returns**

number of bytes needed to store key name including user domain  
1 on empty name  
-1 on NULL pointer

**See also**

[keyGetFullName\(\)](#), [keyGetNameSize\(\)](#)

**7.5.2.7** `ssize_t keyGetName ( const Key * key, char * returnedName, size_t maxSize )`

Get abbreviated key name (without owner name).

When there is not enough space to write the name, nothing will be written and -1 will be returned.

*maxSize* is limited to SSIZE\_MAX. When this value is exceeded -1 will be returned. The reason for that is that any value higher is just a negative return value passed by accident. Of course malloc is not as failure tolerant and will try to allocate.

```
char *getBack = malloc (keyGetNameSize(key));  
keyGetName(key, getBack, keyGetNameSize(key));
```

**Returns**

- number of bytes written to *returnedName*
- 1 when only a null was written
- 1 when keyname is longer than *maxSize* or 0 or any NULL pointer

**Parameters**

- key* the key object to work with
- returnedName* pre-allocated memory to write the key name
- maxSize* maximum number of bytes that will fit in *returnedName*, including the final NULL

**See also**

[keyGetNameSize\(\)](#), [keyGetFullName\(\)](#), [keyGetFullNameSize\(\)](#)

**7.5.2.8** `ssize_t keyGetNameSize ( const Key * key )`

Bytes needed to store the key name without owner.

For an empty key name you need one byte to store the ending NULL. For that reason 1 is returned.

**Parameters**

- key* the key object to work with

**Returns**

- number of bytes needed, including ending NULL, to store key name without owner
- 1 if there is no key Name
- 1 on NULL pointer

**See also**

[keyGetName\(\)](#), [keyGetFullNameSize\(\)](#)

### 7.5.2.9 `ssize_t keyGetOwner ( const Key * key, char * returnedOwner, size_t maxSize )`

Return the owner of the key.

- Given `user:someuser/.....` return `someuser`
- Given `user:some.user/....` return `some.user`
- Given `user/....` return the current user

Only `user/...` keys have a owner. For `system/...` keys (that doesn't have a key owner) an empty string ("") is returned.

Although usually the same, the owner of a key is not related to its UID. Owner are related to WHERE the key is stored on disk, while UIDs are related to mode controls of a key.

#### Parameters

*key* the object to work with

*returnedOwner* a pre-allocated space to store the owner

*maxSize* maximum number of bytes that fit returned

#### Returns

number of bytes written to buffer

1 if there is no owner

-1 on NULL pointers

-1 when `maxSize` is 0, larger than `SSIZE_MAX` or too small for ownername

#### See also

[keySetName\(\)](#), [keySetOwner\(\)](#), [keyOwner\(\)](#), [keyGetFullName\(\)](#)

### 7.5.2.10 `ssize_t keyGetOwnerSize ( const Key * key )`

Return the size of the owner of the Key with concluding 0.

The returned number can be used to allocate a string. 1 will returned on an empty owner to store the concluding 0 on using [keyGetOwner\(\)](#).

```
char * buffer;
buffer = malloc (keyGetOwnerSize (key));
// use buffer and keyGetOwnerSize (key) for maxSize
```

#### Note

that -1 might be returned on null pointer, so when you directly allocate afterwards its best to check if you will pass a null pointer before.

**Parameters**

*key* the key object to work with

**Returns**

number of bytes  
1 if there is no owner  
-1 on NULL pointer

**See also**

[keyGetOwner\(\)](#)

**7.5.2.11 const char\* keyName ( const Key \* *key* )**

Returns a pointer to the abbreviated real internal `key` name.

This is a much more efficient version of [keyGetName\(\)](#) and can use it if you are responsible enough to not mess up things. You are not allowed to change anything in the returned array. The content of that string may change after [keySetName\(\)](#) and similar functions. If you need a copy of the name, consider using [keyGetName\(\)](#).

The name will be without owner, see [keyGetFullName\(\)](#) if you need the name with its owner.

[keyName\(\)](#) returns "" when there is no `keyName`. The reason is

```
key=keyNew(0);  
keySetName(key, "");  
keyName(key); // you would expect "" here  
keyDel(key);
```

**Note**

Note that the `Key` structure keeps its own size field that is calculated by library internal calls, so to avoid inconsistencies, you must never use the pointer returned by [keyName\(\)](#) method to set a new value. Use [keySetName\(\)](#) instead.

**Parameters**

*key* the key object to work with

**Returns**

a pointer to the keyname which must not be changed.  
"" when there is no (a empty) keyname  
0 on NULL pointer

**See also**

[keyGetNameSize\(\)](#) for the string length  
[keyGetFullName\(\)](#), [keyGetFullNameSize\(\)](#) to get the full name  
[keyGetName\(\)](#) as alternative to get a copy  
[keyOwner\(\)](#) to get a pointer to owner

### 7.5.2.12 `const char* keyOwner ( const Key * key )`

Return a pointer to the real internal key owner.

This is a much more efficient version of [keyGetOwner\(\)](#) and you should use it if you are responsible enough to not mess up things. You are not allowed to modify the returned string in any way. If you need a copy of the string, consider to use [keyGetOwner\(\)](#) instead.

[keyOwner\(\)](#) returns "" when there is no keyOwner. The reason is

```
key=keyNew(0);
keySetOwner(key, "");
keyOwner(key); // you would expect "" here
keySetOwner(key, "system");
keyOwner(key); // you would expect "" here
```

#### Note

Note that the Key structure keeps its own size field that is calculated by library internal calls, so to avoid inconsistencies, you must never use the pointer returned by [keyOwner\(\)](#) method to set a new value. Use [keySetOwner\(\)](#) instead.

#### Parameters

*key* the key object to work with

#### Returns

a pointer to internal owner  
 "" when there is no (a empty) owner  
 0 on NULL pointer

#### See also

[keyGetOwnerSize\(\)](#) for the size of the string with concluding 0  
[keyGetOwner\(\)](#), [keySetOwner\(\)](#)  
[keyName\(\)](#) for name without owner  
[keyGetFullName\(\)](#) for name with owner

### 7.5.2.13 `ssize_t keySetBaseName ( Key * key, const char * baseName )`

Sets *baseName* as the new basename for *key*.

All text after the last '/' in the key *keyname* is erased and *baseName* is appended.

So lets suppose *key* has name "system/dir1/dir2/key1". If *baseName* is "key2", the resulting key name will be "system/dir1/dir2/key2". If *baseName* is empty or NULL, the resulting key name will be "system/dir1/dir2".

#### Warning

You should not change a keys name once it belongs to a keyset. See [ksSort\(\)](#) for more information.

TODO: does not work with .. and .

### Parameters

*key* the key object to work with

*baseName* the string used to overwrite the basename of the key

### Returns

the size in bytes of the new key name

-1 on NULL pointers

### See also

[keyAddBaseName\(\)](#)

[keySetName\(\)](#) to set a new name

#### 7.5.2.14 ssize\_t keySetName ( Key \* *key*, const char \* *newName* )

Set a new name to a key.

A valid name is of the forms:

- system/something
- user/something
- user:username/something

The last form has explicitly set the owner, to let the library know in which user folder to save the key. A owner is a user name. If not defined (the second form) current user is calculated and used as default.

You should always follow the guidelines for key tree structure creation.

A private copy of the key name will be stored, and the *newName* parameter can be freed after this call.

.., . and / will be handled correctly. A valid name will be build out of the (valid) name what you pass, e.g. user///sw/./sw//./MyApp -> user/sw/MyApp

On invalid names, NULL or "" the name will be "" afterwards.

### Warning

You should not change a keys name once it belongs to a keyset. See ksSort() for more information.

### Returns

size in bytes of this new key name including ending NULL

-1 if *newName* is empty or invalid or any NULL pointer

**Parameters**

*key* the key object to work with

*newName* the new key name

**See also**

[keyNew\(\)](#), [keySetOwner\(\)](#)  
[keyGetName\(\)](#), [keyGetFullName\(\)](#), [keyName\(\)](#)  
[keySetBaseName\(\)](#), [keyAddBaseName\(\)](#) to manipulate a name

**7.5.2.15 ssize\_t keySetOwner ( Key \* *key*, const char \* *newOwner* )**

Set the owner of a key.

A owner is a name of a system user related to a UID. The owner decides on which location on the disc the key goes.

A private copy is stored, so the passed parameter can be freed after the call.

**Parameters**

*key* the key object to work with

*newOwner* the string which describes the owner of the key

**Returns**

the number of bytes actually saved including final NULL  
 1 when owner is freed (by setting 0 or "")  
 -1 on null pointer or memory problems

**See also**

[keySetName\(\)](#), [keyGetOwner\(\)](#), [keyGetFullName\(\)](#)

**7.6 KeySet :: Class Methods**

Methods to manipulate KeySets.

**Functions**

- [KeySet \\* ksNew](#) (size\_t alloc,...)
- [KeySet \\* ksDup](#) (const [KeySet](#) \*source)
- [int ksCopy](#) ([KeySet](#) \*dest, const [KeySet](#) \*source)
- [int ksDel](#) ([KeySet](#) \*ks)
- [ssize\\_t ksGetSize](#) (const [KeySet](#) \*ks)
- [ssize\\_t ksAppendKey](#) ([KeySet](#) \*ks, [Key](#) \*toAppend)
- [ssize\\_t ksAppend](#) ([KeySet](#) \*ks, const [KeySet](#) \*toAppend)



- [KeySet \\* ksCut](#) ([KeySet \\*ks](#), const [Key \\*cutpoint](#))
- [Key \\* ksPop](#) ([KeySet \\*ks](#))
- [int ksRewind](#) ([KeySet \\*ks](#))
- [Key \\* ksNext](#) ([KeySet \\*ks](#))
- [Key \\* ksCurrent](#) (const [KeySet \\*ks](#))
- [Key \\* ksHead](#) (const [KeySet \\*ks](#))
- [Key \\* ksTail](#) (const [KeySet \\*ks](#))
- [cursor\\_t ksGetCursor](#) (const [KeySet \\*ks](#))
- [int ksSetCursor](#) ([KeySet \\*ks](#), [cursor\\_t cursor](#))
- [Key \\* ksLookup](#) ([KeySet \\*ks](#), [Key \\*key](#), [option\\_t options](#))
- [Key \\* ksLookupByName](#) ([KeySet \\*ks](#), const char \*name, [option\\_t options](#))

### 7.6.1 Detailed Description

Methods to manipulate KeySets. A KeySet is a sorted set of keys. So the correct name actually would be KeyMap.

Terminate with [ksNew\(0\)](#) or [ksNew\(20, ..., KS\\_END\)](#) This is because there is a list of [Key\\*](#) required and [KS\\_END](#) has the length of ([Key\\*](#)).

It can be implemented in various ways like a linked list or with a dynamically allocated array.

With [ksNew\(\)](#) you can create a new KeySet.

You can add keys with [ksAppendKey\(\)](#) in the keyset. [ksGetSize\(\)](#) tells you the current size of the keyset.

With [ksRewind\(\)](#) and [ksNext\(\)](#) you can navigate through the keyset. Don't expect any particular order, but it is assured that you will get every key of the set.

KeySets have an [internal cursor](#) . This is used for [ksLookup\(\)](#) and [kdbSet\(\)](#).

KeySet has a fundamental meaning inside elektra. It makes it possible to get and store many keys at once inside the database. In addition to that the class can be used as high level datastructure in applications. With [ksLookupByName\(\)](#) it is possible to fetch easily specific keys out of the list of keys.

You can easily create and iterate keys:

```
#include <kdb.h>

// create a new keyset with 3 keys
// with a hint that about 20 keys will be inside
KeySet *myConfig = ksNew(20,
    keyNew ("user/name1", 0),
    keyNew ("user/name2", 0),
    keyNew ("user/name3", 0),
    KS_END);
// append a key in the keyset
ksAppendKey(myConfig, keyNew("user/name4", 0));

Key *current;
ksRewind(myConfig);
while ((current=ksNext(myConfig))!=0)
{
```

```

        printf("Key name is %s.\n", keyName (current));
    }
    ksDel (myConfig); // delete keyset and all keys appended

```

## 7.6.2 Function Documentation

### 7.6.2.1 `ssize_t ksAppend ( KeySet * ks, const KeySet * toAppend )`

Append all `toAppend` contained keys to the end of the `ks`.

`toAppend` `KeySet` will be left unchanged.

If a key is both in `toAppend` and `ks`, the `Key` in `ks` will be overridden.

#### Postcondition

Sorted `KeySet` `ks` with all keys it had before and additionally the keys from `toAppend`

#### Returns

the size of the `KeySet` after transfer  
-1 on NULL pointers

#### Parameters

*ks* the `KeySet` that will receive the keys  
*toAppend* the `KeySet` that provides the keys that will be transferred

#### See also

[ksAppendKey\(\)](#)

### 7.6.2.2 `ssize_t ksAppendKey ( KeySet * ks, Key * toAppend )`

Appends a `Key` to the end of `ks`.

A pointer to the key will be stored, and not a private copy. So a future [ksDel\(\)](#) on `ks` may [keyDel\(\)](#) the `toAppend` object, see [keyGetRef\(\)](#).

The reference counter of the key will be incremented, and thus `toAppend` is not const.

If the keyname already existed, it will be replaced with the new key.

The `KeySet` internal cursor will be set to the new key.

#### Returns

the size of the `KeySet` after insertion  
-1 on NULL pointers  
-1 if insertion failed, the key will be deleted then.

**Parameters**

*ks* KeySet that will receive the key

*toAppend* Key that will be appended to *ks*

**See also**

[ksAppend\(\)](#), [keyNew\(\)](#), [ksDel\(\)](#)  
[keyIncRef\(\)](#)

**7.6.2.3 int ksCopy ( KeySet \* *dest*, const KeySet \* *source* )**

Copy a keyset.

Most often you may want a duplicate of a keyset, see [ksDup\(\)](#) or append keys, see [ksAppend\(\)](#). But in some situations you need to copy a keyset to a existing keyset, for that this function exists.

You can also use it to clear a keyset when you pass a NULL pointer as *source*.

Note that all keys in *dest* will be deleted. Afterwards the content of the source will be added to the destination and the [ksCurrent\(\)](#) is set properly in *dest*.

A flat copy is made, so the keys will not be duplicated, but there reference counter is updated, so both keysets need to be [ksDel\(\)](#).

```
int f (KeySet *ks)
{
    KeySet *c = ksNew (20, ..., KS_END);
    // c receives keys
    ksCopy (ks, c); // pass the keyset to the caller

    ksDel (c);
    // caller needs to ksDel (ks)
}
```

**Parameters**

*source* has to be an initialized source KeySet or NULL

*dest* has to be an initialized KeySet where to write the keys

**Returns**

1 on success

0 if *dest* was cleared successfully (*source* is NULL)

-1 on NULL pointer

**See also**

[ksNew\(\)](#), [ksDel\(\)](#), [ksDup\(\)](#)  
[keyCopy\(\)](#) for copying keys

#### 7.6.2.4 Key\* ksCurrent ( const KeySet \* ks )

Return the current Key.

The pointer is NULL if you reached the end or after [ksRewind\(\)](#).

##### Note

You must not delete the key or change the key, use [ksPop\(\)](#) if you want to delete it.

##### Parameters

*ks* the keyset object to work with

##### Returns

pointer to the Key pointed by *ks*'s cursor  
0 on NULL pointer

##### See also

[ksNext\(\)](#), [ksRewind\(\)](#)

#### 7.6.2.5 KeySet\* ksCut ( KeySet \* ks, const Key \* cutpoint )

Cuts out a keyset at the cutpoint.

Searches for the cutpoint inside the KeySet *ks*. If found it cuts out everything which is below (see [keyIsBelow\(\)](#)) this key. If not found an empty keyset is returned.

The cursor will stay at the same key as it was before. If the cursor was inside the region of cutted (moved) keys, the cursor will be set to the key before the cutpoint.

##### Returns

a new allocated KeySet which needs to be deleted with [ksDel\(\)](#). The keyset consists of all keys (of the original keyset *ks*) below the cutpoint. If the key cutpoint exists, it will also be appended.

##### Return values

0 on null pointers, no key name or allocation problems

##### Parameters

*ks* the keyset to cut. It will be modified by removing all keys below the cutpoint. The cutpoint itself will also be removed.

*cutpoint* the point where to cut out the keyset

### 7.6.2.6 int ksDel ( KeySet \* *ks* )

A destructor for KeySet objects.

Cleans all internal dynamic attributes, decrement all reference pointers to all keys and then [keyDel\(\)](#) all contained Keys, and free(s) the release the KeySet object memory (that was previously allocated by [ksNew\(\)](#)).

#### Parameters

*ks* the keyset object to work with

#### Returns

0 when the keyset was freed  
-1 on null pointer

#### See also

[ksNew\(\)](#)

### 7.6.2.7 KeySet\* ksDup ( const KeySet \* *source* )

Return a duplicate of a keyset.

Objects created with [ksDup\(\)](#) must be destroyed with [ksDel\(\)](#).

Memory will be allocated as needed for dynamic properties, so you need to [ksDel\(\)](#) the returned pointer.

A flat copy is made, so the keys will not be duplicated, but there reference counter is updated, so both keysets need [ksDel\(\)](#).

#### Parameters

*source* has to be an initialised source KeySet

#### Returns

a flat copy of source on success  
0 on NULL pointer

#### See also

[ksNew\(\)](#), [ksDel\(\)](#)  
[keyDup\(\)](#) for [Key :: Basic Methods](#) duplication

### 7.6.2.8 cursor\_t ksGetCursor ( const KeySet \* *ks* )

Get the KeySet internal cursor.

Use it to get the cursor of the actual position.

### Warning

Cursors are getting invalid when the key was [ksPop\(\)](#) or [ksLookup\(\)](#) with KDB\_O\_POP was used.

### 7.6.3 Read ahead

With the cursors it is possible to read ahead in a keyset:

```
cursor_t jump;
ksRewind (ks);
while ((key = keyNextMeta (ks))!=0)
{
    // now mark this key
    jump = ksGetCursor(ks);

    //code..
    keyNextMeta (ks); // now browse on
    // use ksCurrent(ks) to check the keys
    //code..

    // jump back to the position marked before
    ksSetCursor(ks, jump);
}
```

### 7.6.4 Restoring state

It can also be used to restore the state of a keyset in a function

```
int f (KeySet *ks)
{
    cursor_t state = ksGetCursor(ks);

    // work with keyset

    // now bring the keyset to the state before
    ksSetCursor (ks, state);
}
```

It is of course possible to make the KeySet const and cast its const away to set the cursor. Another way to achieve the same is to [ksDup\(\)](#) the keyset, but it is not as efficient.

An invalid cursor will be returned directly after [ksRewind\(\)](#). When you set an invalid cursor [ksCurrent\(\)](#) is 0 and [ksNext\(\) == ksHead\(\)](#).

### Note

Only use a cursor for the same keyset which it was made for.

### Parameters

*ks* the keyset object to work with

**Returns**

a valid cursor on success  
an invalid cursor on NULL pointer or after [ksRewind\(\)](#)

**See also**

[ksNext\(\)](#), [ksSetCursor\(\)](#)

**7.6.4.1 ssize\_t ksGetSize ( const KeySet \* ks )**

Return the number of keys that *ks* contains.

**Parameters**

*ks* the keyset object to work with

**Returns**

the number of keys that *ks* contains.  
-1 on NULL pointer

**See also**

[ksNew\(0\)](#), [ksDel\(\)](#)

**7.6.4.2 Key\* ksHead ( const KeySet \* ks )**

Return the first key in the KeySet.

The KeySets cursor will not be effected.

If [ksCurrent\(\)==ksHead\(\)](#) you know you are on the first key.

**Parameters**

*ks* the keyset object to work with

**Returns**

the first Key of a keyset  
0 on NULL pointer or empty keyset

**See also**

[ksTail\(\)](#) for the last [Key :: Basic Methods](#)  
[ksRewind\(\)](#), [ksCurrent\(\)](#) and [ksNext\(\)](#) for iterating over the [KeySet :: Class Methods](#)

**7.6.4.3 Key\* ksLookup ( KeySet \* ks, Key \* key, option\_t options )**

Look for a Key contained in *ks* that matches the name of the *key*.

### 7.6.5 Introduction

`ksLookup()` is designed to let you work with entirely pre-loaded KeySets, so instead of `kdbGetKey()`, key by key, the idea is to fully `kdbGet()` for your application root key and process it all at once with `ksLookup()`.

This function is very efficient by using binary search. Together with `kdbGet()` which can you load the whole configuration with only some communication to backends you can write very effective but short code for configuration.

### 7.6.6 Usage

If found, `ks` internal cursor will be positioned in the matched key (also accessible by `ksCurrent()`), and a pointer to the Key is returned. If not found, `ks` internal cursor will not move, and a NULL pointer is returned.

Cascading is done if the first character is a `/`. This leads to ignoring the prefix like `system/` and `user/`.

```
if (kdbGet(handle, "user/myapp", myConfig, 0) == -1)
    errorHandler ("Could not get Keys");

if (kdbGet(handle, "system/myapp", myConfig, 0) == -1)
    errorHandler ("Could not get Keys");

if ((myKey = ksLookup(myConfig, key, 0)) == NULL)
    errorHandler ("Could not Lookup Key");
```

This is the way multi user Programs should get there configuration and search after the values. It is guaranteed that more namespaces can be added easily and that all values can be set by admin and user.

#### 7.6.6.1 KDB\_O\_NOALL

When `KDB_O_NOALL` is set the keyset will be only searched from `ksCurrent()` to `ksTail()`. You need to `ksRewind()` the keyset yourself. `ksCurrent()` is always set properly after searching a key, so you can go on searching another key after the found key.

When `KDB_O_NOALL` is not set the cursor will stay untouched and all keys are considered. A much more efficient binary search will be used then.

#### 7.6.6.2 KDB\_O\_POP

When `KDB_O_POP` is set the key which was found will be `ksPop()`ed. `ksCurrent()` will not be changed, only iff `ksCurrent()` is the searched key, then the keyset will be `ksRewind()`ed.

#### Note

Like in `ksPop()` the popped key always needs to be `keyDel()` afterwards, even if it is appended to another keyset.



**Warning**

All cursors on the keyset will be invalid iff you use KDB\_O\_POP, so don't use this if you rely on a cursor, see [ksGetCursor\(\)](#).

You can solve this problem by using KDB\_O\_NOALL, risking you have to iterate  $n^2$  instead of  $n$ .

The more elegant way is to separate the keyset you use for [ksLookup\(\)](#) and [ksAppendKey\(\)](#):

```
int f(KeySet *iterator, KeySet *lookup)
{
    KeySet *append = ksNew (ksGetSize(lookup), KS_END);
    Key *key;
    Key *current;

    ksRewind(iterator);
    while (current=ksNext(iterator))
    {
        key = ksLookup (lookup, current, KDB_O_POP);
        // do something...
        ksAppendKey(append, key); // now append it to append, not lookup!

        keyDel (key); // make sure to ALWAYS delete popped keys.
    }
    ksAppend(lookup, append);
    // now lookup needs to be sorted only once, append never
    ksDel (append);
}
```

**Parameters**

**ks** where to look for

**key** the key object you are looking for

**options** some KDB\_O\_\* option bits:

- KDB\_O\_NOCASE  
Lookup ignoring case.
- KDB\_O\_WITHOWNER  
Also consider correct owner.
- KDB\_O\_NOALL  
Only search from [ksCurrent\(\)](#) to end of keyset, see above text.
- KDB\_O\_POP  
Pop the key which was found.
- KDB\_O\_DEL  
Delete the passed key.

**Returns**

pointer to the Key found, 0 otherwise  
0 on NULL pointers

**See also**

[ksLookupByName\(\)](#) to search by a name given by a string

[ksCurrent\(\)](#), [ksRewind\(\)](#), [ksNext\(\)](#) for iterating over a [KeySet :: Class Methods](#)

### 7.6.6.3 Key\* ksLookupByName ( KeySet \* ks, const char \* name, option\_t options )

Look for a Key contained in ks that matches name.

[ksLookupByName\(\)](#) is designed to let you work with entirely pre-loaded KeySets, so instead of [kdbGetKey\(\)](#), key by key, the idea is to fully [kdbGetByName\(\)](#) for your application root key and process it all at once with [ksLookupByName\(\)](#).

This function is very efficient by using binary search. Together with [kdbGetByName\(\)](#) which can you load the whole configuration with only some communication to back-ends you can write very effective but short code for configuration.

If found, ks internal cursor will be positioned in the matched key (also accessible by [ksCurrent\(\)](#)), and a pointer to the Key is returned. If not found, ks internal cursor will not move, and a NULL pointer is returned. If requested to pop the key, the cursor will be rewinded.

## 7.6.7 Cascading

Cascading is done if the first character is a /. This leads to ignoring the prefix like system/ and user/.

```
if (kdbGet(handle, "user/sw/myapp/current", myConfig, parentKey) == -1)
    errorHandler ("Could not get Keys", parentKey);

if (kdbGet(handle, "system/sw/myapp/current", myConfig, parentKey) == -1)
    errorHandler ("Could not get Keys", parentKey);

if ((myKey = ksLookupByName (myConfig, "/myapp/current/key", 0)) == NULL)
    errorHandler ("Could not Lookup Key");
```

This is the way multi user Programs should get there configuration and search after the values. It is guaranteed that more namespaces can be added easily and that all values can be set by admin and user.

It is up to the application to implement a sophisticated cascading algorithm, for e.g. a list of profiles (specific, group and fallback):

```
if ((myKey = ksLookupByName (myConfig, "/myapp/current/specific/key", 0)) == NULL)
    if ((myKey = ksLookupByName (myConfig, "/myapp/current/group/key", 0)) == NULL)
        if ((myKey = ksLookupByName (myConfig, "/myapp/current/fallback/key", 0)) == NULL)
            errorHandler ("All fallbacks failed to lookup key");
```

Note that for every profile both the user and the system key are searched. The first key found will be used.

## 7.6.8 Full Search

When KDB\_O\_NOALL is set the keyset will be only searched from [ksCurrent\(\)](#) to [ksTail\(\)](#). You need to [ksRewind\(\)](#) the keyset yourself. [ksCurrent\(\)](#) is always set properly after searching a key, so you can go on searching another key after the found key.

When KDB\_O\_NOALL is not set the cursor will stay untouched and all keys are considered. A much more efficient binary search will be used then.

### Parameters

*ks* where to look for

*name* key name you are looking for

*options* some KDB\_O\_\* option bits:

- KDB\_O\_NOCASE  
Lookup ignoring case.
- KDB\_O\_WITHOWNER  
Also consider correct owner.
- KDB\_O\_NOALL  
Only search from [ksCurrent\(\)](#) to end of keyset, see above text.
- KDB\_O\_POP  
Pop the key which was found.

Currently no options supported.

### Returns

pointer to the Key found, 0 otherwise  
0 on NULL pointers

### See also

[keyCompare\(\)](#) for very powerfull Key lookups in KeySets  
[ksCurrent\(\)](#), [ksRewind\(\)](#), [ksNext\(\)](#)

#### 7.6.8.1 KeySet\* ksNew ( size\_t alloc, ... )

Allocate, initialize and return a new KeySet object.

Objects created with [ksNew\(\)](#) must be destroyed with [ksDel\(\)](#).

You can use a various long list of parameters to preload the keyset with a list of keys. Either your first and only parameter is 0 or your last parameter must be KEY\_END.

For most uses

```
KeySet *keys = ksNew(0);
// work with it
ksDel (keys);
```

goes ok, the alloc size will be 16, defined in kdbprivate.h. The alloc size will be doubled whenever size reaches alloc size, so it also performs out large keysets.

But if you have any clue how large your keyset may be you should read the next statements.

If you want a keyset with length 15 (because you know of your application that you normally need about 12 up to 15 keys), use:

```

KeySet * keys = ksNew (15,
    keyNew ("user/sw/app/fixedConfiguration/key01", KEY_SWITCH_VALUE, "value0
1", 0),
    keyNew ("user/sw/app/fixedConfiguration/key02", KEY_SWITCH_VALUE, "value0
2", 0),
    keyNew ("user/sw/app/fixedConfiguration/key03", KEY_SWITCH_VALUE, "value0
3", 0),
    // ...
    keyNew ("user/sw/app/fixedConfiguration/key15", KEY_SWITCH_VALUE, "value1
5", 0),
    KS_END);
// work with it
ksDel (keys);

```

If you start having 3 keys, and your application needs approximately 200-500 keys, you can use:

```

KeySet * config = ksNew (500,
    keyNew ("user/sw/app/fixedConfiguration/key1", KEY_SWITCH_VALUE, "value1"
, 0),
    keyNew ("user/sw/app/fixedConfiguration/key2", KEY_SWITCH_VALUE, "value2"
, 0),
    keyNew ("user/sw/app/fixedConfiguration/key3", KEY_SWITCH_VALUE, "value3"
, 0),
    KS_END); // don't forget the KS_END at the end!
// work with it
ksDel (config);

```

Alloc size is 500, the size of the keyset will be 3 after ksNew. This means the keyset will reallocate when appending more than 497 keys.

The main benefit of taking a list of variant length parameters is to be able to have one C-Statement for any possible KeySet.

Due to ABI compatibility, the KeySet structure is only declared in kdb.h, and not defined. So you can only declare pointers to KeySets in your program. See <http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html#AEN135>

#### See also

[ksDel\(\)](#) to free the [KeySet :: Class Methods](#) afterwards  
[ksDup\(\)](#) to duplicate an existing [KeySet :: Class Methods](#)

#### Parameters

*alloc* gives a hint for the size how many Keys may be stored initially

#### Returns

a ready to use KeySet object  
0 on memory error

#### 7.6.8.2 Key\* ksNext ( KeySet \* ks )

Returns the next Key in a KeySet.

KeySets have an internal cursor that can be reset with [ksRewind\(\)](#). Every time [ksNext\(\)](#) is called the cursor is incremented and the new current Key is returned.

You'll get a NULL pointer if the key after the end of the KeySet was reached. On subsequent calls of [ksNext\(\)](#) it will still return the NULL pointer.

The `ks` internal cursor will be changed, so it is not const.

#### Note

You must not delete or change the key, use [ksPop\(\)](#) if you want to delete it.

#### Parameters

*ks* the keyset object to work with

#### Returns

the new current Key  
0 when the end is reached  
0 on NULL pointer

#### See also

[ksRewind\(\)](#), [ksCurrent\(\)](#)

#### 7.6.8.3 Key\* ksPop ( KeySet \* ks )

Remove and return the last key of `ks`.

The reference counter will be decremented by one.

The KeySets cursor will not be effected if it did not point to the popped key.

#### Note

You need to [keyDel\(\)](#) the key afterwards, if you don't append it to another keyset. It has the same semantics like a key allocated with [keyNew\(\)](#) or [keyDup\(\)](#).

```
ks1=ksNew(0);
ks2=ksNew(0);

k1=keyNew("user/name", KEY_END); // ref counter 0
ksAppendKey(ks1, k1); // ref counter 1
ksAppendKey(ks2, k1); // ref counter 2

k1=ksPop(ks1); // ref counter 1
k1=ksPop(ks2); // ref counter 0, like after keyNew()

ksAppendKey(ks1, k1); // ref counter 1

ksDel(ks1); // key is deleted too
ksDel(ks2);
*
```

**Returns**

the last key of `ks`  
 NULL if `ks` is empty or on NULL pointer

**Parameters**

*ks* KeySet to work with

**See also**

[ksAppendKey\(\)](#), [ksAppend\(\)](#)  
[commandList\(\)](#) for an example

**7.6.8.4 int ksRewind ( KeySet \* ks )**

Rewinds the KeySet internal cursor.

Use it to set the cursor to the beginning of the KeySet. [ksCurrent\(\)](#) will then always return NULL afterwards. So you want to [ksNext\(\)](#) first.

```
ksRewind (ks);
while ((key = ksNext (ks)) != 0) {}
```

**Parameters**

*ks* the keyset object to work with

**Returns**

0 on success  
 -1 on NULL pointer

**See also**

[ksNext\(\)](#), [ksCurrent\(\)](#)

**7.6.8.5 int ksSetCursor ( KeySet \* ks, cursor\_t cursor )**

Set the KeySet internal cursor.

Use it to set the cursor to a stored position. [ksCurrent\(\)](#) will then be the position which you got with.

**Warning**

Cursors may get invalid when the key was [ksPop\(\)](#)ed or [ksLookup\(\)](#) was used together with KDB\_O\_POP.

```
cursor_t cursor;
..
```

```
// key now in any position here
cursor = ksGetCursor (ks);
while ((key = keyNextMeta (ks))!=0) {}
ksSetCursor (ks, cursor); // reset state
ksCurrent(ks); // in same position as before
```

An invalid cursor will set the keyset to its beginning like [ksRewind\(\)](#). When you set an invalid cursor [ksCurrent\(\)](#) is 0 and [ksNext\(\)](#) == [ksHead\(\)](#).

#### Parameters

*cursor* the cursor to use  
*ks* the keyset object to work with

#### Returns

0 when the keyset is [ksRewind\(\)](#)ed  
 1 otherwise  
 -1 on NULL pointer

#### See also

[ksNext\(\)](#), [ksGetCursor\(\)](#)

#### 7.6.8.6 Key\* ksTail ( const KeySet \* ks )

Return the last key in the KeySet.

The KeySets cursor will not be effected.

If [ksCurrent\(\)](#)==[ksTail\(\)](#) you know you are on the last key. [ksNext\(\)](#) will return a NULL pointer afterwards.

#### Parameters

*ks* the keyset object to work with

#### Returns

the last Key of a keyset  
 0 on NULL pointer or empty keyset

#### See also

[ksHead\(\)](#) for the first [Key :: Basic Methods](#)  
[ksRewind\(\)](#), [ksCurrent\(\)](#) and [ksNext\(\)](#) for iterating over the [KeySet :: Class Methods](#)

## 7.7 Key :: Methods for Making Tests

Methods to do various tests on Keys.

## Functions

- `int keyCmp (const Key *k1, const Key *k2)`
- `int keyClearSync (Key *key)`
- `int keyNeedSync (const Key *key)`
- `int keyIsSystem (const Key *key)`
- `int keyIsUser (const Key *key)`
- `int keyIsBelow (const Key *key, const Key *check)`
- `int keyIsDirectBelow (const Key *key, const Key *check)`
- `int keyRel (const Key *key, const Key *check)`
- `int keyIsInactive (const Key *key)`
- `int keyIsDir (const Key *key)`
- `int keyIsBinary (const Key *key)`
- `int keyIsString (const Key *key)`
- `keyswitch_t keyCompare (const Key *key1, const Key *key2)`

### 7.7.1 Detailed Description

Methods to do various tests on Keys. To use them:

```
#include <kdb.h>
```

### 7.7.2 Function Documentation

#### 7.7.2.1 `int keyClearSync ( Key * key )`

Clear flags of a key.

#### Todo

Should be done only in `kdbGet()` part of plugins.

If you want to get the current flags, just call it with `semiflag` set to 0.

#### Parameters

*key* the key object to work with

#### Returns

-1 on null key  
new flags for that key otherwise



**7.7.2.2 int keyCmp ( const Key \* k1, const Key \* k2 )**

Compare two keys.

**Returns**

a number less than, equal to or greater than zero if k1 is found, respectively, to be less than, to match, or be greater than k2.

The comparison is based on a strcmp of the keynames, and iff they match a strcmp of the owner will be used to distinguish. If even this matches the keys are found to be exactly the same and 0 is returned. These two keys can't be used in the same KeySet.

[keyCmp\(\)](#) defines the sorting order for a KeySet.

The following 3 points are the rules for null values. They only take account when none of the preceding rules matched.

- A null pointer will be found to be smaller than every other key. If both are null pointers, 0 is returned.
- A null name will be found to be smaller than every other name. If both are null names, 0 is returned.
- No owner will be found to be smaller than every other owner. If both don't have a owner, 0 is returned.

**Note**

the owner will only be used if the names are equal.

Often is enough to know if the other key is less then or greater then the other one. But Sometimes you need more precise information, see [keyRel\(\)](#).

Given any Keys k1 and k2 constructed with [keyNew\(\)](#), following equation hold true:

```
// keyCmp(0,0) == 0
// keyCmp(k1,0) == 1
// keyCmp(0,k2) == -1
```

You can write similar equation for the other rules.

Here are some more examples with equation:

```
Key *k1 = keyNew("user/a", KEY_END);
Key *k2 = keyNew("user/b", KEY_END);

// keyCmp(k1,k2) < 0
// keyCmp(k2,k1) > 0

Key *k1 = keyNew("user/a", KEY_OWNER, "markus", KEY_END);
Key *k2 = keyNew("user/a", KEY_OWNER, "max", KEY_END);

// keyCmp(k1,k2) < 0
// keyCmp(k2,k1) > 0
```

**Warning**

Do not try to strcmp the [keyName\(\)](#) yourself because the used strcmp implementation is allowed to differ from simple ascii comparison.

**Parameters**

*k1* the first key object to compare with  
*k2* the second key object to compare with

**See also**

[ksAppendKey\(\)](#), [ksAppend\(\)](#) will compare keys when appending  
[ksLookup\(\)](#) will compare keys during searching

**7.7.2.3 keyswitch\_t keyCompare ( const Key \* *key1*, const Key \* *key2* )**

Compare 2 keys.

The returned flags bit array has 1s (differ) or 0s (equal) for each key meta info compared, that can be logically ORed using [keyswitch\\_t](#) flags. [KEY\\_NAME](#) , [KEY\\_VALUE](#) , [KEY\\_OWNER](#) , [KEY\\_COMMENT](#) , [KEY\\_UID](#) , [KEY\\_GID](#) , [KEY\\_MODE](#) and

**A very simple example would be**

```
Key *key1, *key;
uint32_t changes;

// omitted key1 and key2 initialization and manipulation

changes=keyCompare(key1,key2);

if (changes == 0) printf("key1 and key2 are identical\n");

if (changes & KEY_VALUE)
    printf("key1 and key2 have different values\n");

if (changes & KEY_UID)
    printf("key1 and key2 have different UID\n");
```

**Example of very powerfull specific Key lookup in a KeySet:**

```
KDB *handle = kdbOpen();
KeySet *ks=ksNew(0);
Key *base = keyNew ("user/sw/MyApp/something", KEY_END);
Key *current;
uint32_t match;
uint32_t interests;

kdbGetByName(handle, ks, "user/sw/MyApp", 0);

// we are interested only in key type and access permissions
interests=(KEY_TYPE | KEY_MODE);
```

```

ksRewind(ks);    // put cursor in the begining
while ((current=ksNext(ks)) {
    match=keyCompare(current,base);

    if ((~match & interests) == interests)
        printf("Key %s has same type and permissions of base key",
            keyName(current));

    // continue walking in the KeySet....
}

// now we want same name and/or value
interests=(KEY_NAME | KEY_VALUE);

// we don't really need ksRewind(), since previous loop achieved end of KeySet
ksRewind(ks);
while ((current=ksNext(ks)) {
    match=keyCompare(current,base);

    if ((~match & interests) == interests) {
        printf("Key %s has same name, value, and sync status
            of base key",keyName(current));
    }
    // continue walking in the KeySet....
}

keyDel(base);
ksDel(ks);
kdbClose(handle);

```

### Returns

a bit array pointing the differences

### Parameters

**key1** first key

**key2** second key

### See also

[keyswitch\\_t](#)

#### 7.7.2.4 int keyIsBelow ( const Key \* *key*, const Key \* *check* )

Check if the key check is below the key key or not.

Example:

```

key user/sw/app
check user/sw/app/key

```

returns true because check is below key

Example:

```

key user/sw/app
check user/sw/app/folder/key

```

returns also true because check is indirect below key

#### Parameters

*key* the key object to work with  
*check* the key to find the relative position of

#### Returns

1 if check is below key  
0 if it is not below or if it is the same key

#### See also

[keySetName\(\)](#), [keyGetName\(\)](#), [keyIsDirectBelow\(\)](#)

#### 7.7.2.5 int keyIsBinary ( const Key \* key )

Check if a key is binary type.

The function checks if the key is a binary. Opposed to string values binary values can have '\0' inside the value and may not be terminated by a null character. Their disadvantage is that you need to pass their size.

Make sure to use this function and don't test the binary type another way to ensure compatibility and to write less error prone programs.

#### Returns

1 if it is binary  
0 if it is not  
-1 on NULL pointer

#### See also

[keyGetBinary\(\)](#), [keySetBinary\(\)](#)

#### Parameters

*key* the key to check

#### 7.7.2.6 int keyIsDir ( const Key \* key )

Check if the mode for the key has access privileges.

In the filesystem backend a key represented through a file has the mode 664, but a key represented through a folder 775. [keyIsDir\(\)](#) checks if all 3 executeable bits are set.

If any executable bit is set it will be recognized as a directory.

#### Note

[keyIsDir](#) may return true even though you can't access the directory.

To know if you can access the directory, you need to check, if your

- user ID is equal the key's user ID and the mode & 100 is true
- group ID is equal the key's group ID and the mode & 010 is true
- mode & 001 is true

Accessing does not mean that you can get any value or comments below, see [Modes](#) for more information.

#### Note

currently mountpoints can only where [keyIsDir\(\)](#) is true (0.7.0) but this is likely to change.

#### Parameters

*key* the key object to work with

#### Returns

1 if key is a directory, 0 otherwise  
-1 on NULL pointer

#### See also

[keySetDir\(\)](#), [keySetMode\(\)](#)

#### 7.7.2.7 int keyIsDirectBelow ( const Key \* *key*, const Key \* *check* )

Check if the key check is direct below the key key or not.

Example:

```
key user/sw/app
check user/sw/app/key
```

returns true because check is below key

Example:

```
key user/sw/app
check user/sw/app/folder/key
```

does not return true, because there is only a indirect relation

#### Parameters

*key* the key object to work with

*check* the key to find the relative position of

#### Returns

1 if check is below key  
0 if it is not below or if it is the same key  
-1 on null pointer

**See also**

[keyIsBelow\(\)](#), [keySetName\(\)](#), [keyGetName\(\)](#)

**7.7.2.8 int keyIsInactive ( const Key \* key )**

Check whether a key is inactive or not.

In elektra terminology any key is inactive if the its basename starts with '.'. Inactive keys must not have any meaning to applications, they are reserved for users and administrators.

To remove a whole hierarchy in elektra, don't forget to pass `option_t::KDB_O_INACTIVE` to [kdbGet\(\)](#) to receive the inactive keys in order to remove them.

Otherwise you should not fetch these keys.

**Parameters**

*key* the key object to work with

**Returns**

1 if the key is inactive, 0 otherwise  
-1 on NULL pointer or when key has no name

**7.7.2.9 int keyIsString ( const Key \* key )**

Check if a key is string type.

String values are null terminated and are not allowed to have any '\0' characters inside the string.

Make sure to use this function and don't test the string type another way to ensure compatibility and to write less error prone programs.

**Returns**

1 if it is string  
0 if it is not  
-1 on NULL pointer

**See also**

[keyGetString\(\)](#), [keySetString\(\)](#)

**Parameters**

*key* the key to check

**7.7.2.10 int keyIsSystem ( const Key \* key )**

Check whether a key is under the `system` namespace or not

**Parameters**

*key* the key object to work with

**Returns**

1 if key name begins with `system`, 0 otherwise  
-1 on NULL pointer

**See also**

[keyIsUser\(\)](#), [keySetName\(\)](#), [keyName\(\)](#)

**7.7.2.11 int keyIsUser ( const Key \* key )**

Check whether a key is under the `user` namespace or not.

**Parameters**

*key* the key object to work with

**Returns**

1 if key name begins with `user`, 0 otherwise  
-1 on NULL pointer

**See also**

[keyIsSystem\(\)](#), [keySetName\(\)](#), [keyName\(\)](#)

**7.7.2.12 int keyNeedSync ( const Key \* key )**

Test if a key needs to be synced to backend storage.

If any key modification took place the key will be flagged with `KEY_FLAG_SYNC` so that [kdbSet\(\)](#) knows which keys were modified and which not.

After [keyNew\(\)](#) the flag will normally be set, but after [kdbGet\(\)](#) and [kdbSet\(\)](#) the flag will be removed. When you modify the key the flag will be set again.

In your application you can make use of that flag to know if you changed something in a key after a [kdbGet\(\)](#) or [kdbSet\(\)](#).

**Note**

Note that also changes in the meta data will set that flag.

**See also**[keyNew\(\)](#)**Parameters***key* the key object to work with**Returns**

1 if *key* was changed in memory, 0 otherwise  
 -1 on NULL pointer

**7.7.2.13 int keyRel ( const Key \* *key*, const Key \* *check* )**

Information about the relation in the hierarchy between two keys.

Unlike [keyCmp\(\)](#) the number gives information about hierarchical information.

- If the keys are the same 0 is returned. So it is the key itself.

```
user/key
user/key
```

```
keySetName (key, "user/key/folder");
keySetName (check, "user/key/folder");
succeed_if (keyRel (key, check) == 0, "should be same");
*
```

**Note**

this relation can be checked with [keyCmp\(\)](#) too.

- If the key is direct below the other one 1 is returned. That means that, in terms of hierarchy, no other key is between them - it is a direct child.

```
user/key/folder
user/key/folder/child
```

```
keySetName (key, "user/key/folder");
keySetName (check, "user/key/folder/child");
succeed_if (keyRel (key, check) == 1, "should be direct below");
*
```

- If the key is below the other one, but not directly 2 is returned. This is also called grand-child.

```
user/key/folder
user/key/folder/any/depth/deeper/grand-child
```

```
keySetName (key, "user/key/folder");
keySetName (check, "user/key/folder/any/depth/deeper/grand-child");
succeed_if (keyRel (key, check) >= 2, "should be below (but not direct)");
succeed_if (keyRel (key, check) > 0, "should be below");
succeed_if (keyRel (key, check) >= 0, "should be the same or below");
*
```



- If a invalid or null ptr key is passed, -1 is returned
- If the keys have no relations, but are not invalid, -2 is returned.
- If the keys are in the same hierarchy, a value smaller then -2 is returned. It means that the key is not below.

```
user/key/myself
user/key/sibling
```

```
keySetName (key, "user/key/folder");
keySetName (check, "user/notsame/folder");
succeed_if (keyRel (key, check) < -2, "key is not below, but same namespace");
```

TODO Below is an idea how it could be extended: It could continue the search into the other direction if any (grand-)parents are equal.

- If the keys are direct below a key which is next to the key, -2 is returned. This is also called nephew. (TODO not implemented)

```
user/key/myself
user/key/sibling
```

- If the keys are direct below a key which is next to the key, -2 is returned. This is also called nephew. (TODO not implemented)

```
user/key/myself
user/key/sibling/nephew
```

- If the keys are below a key which is next to the key, -3 is returned. This is also called grand-nephew. (TODO not implemented)

```
user/key/myself
user/key/sibling/any/depth/deeper/grand-nephew
```

The same holds true for the other direction, but with negative values. For no relation INT\_MIN is returned.

### Note

to check if the keys are the same, you must use `keyCmp() == 0`! `keyRel()` does not give you the information if it did not find a relation or if it is the same key.

### Returns

dependend on the relation: 2.. if below 1.. if direct below 0.. if the same -1.. on null or invalid keys -2.. if none of any other relation -3.. if same hierarchy (none of those below) -4.. if sibling (in same hierarchy) -5.. if nephew (in same hierarchy)

### Parameters

**key** the key object to work with

**check** the second key object to check the relation with

## 7.8 Key :: Value Manipulation Methods

Methods to do various operations on Key values.

### Functions

- `const void * keyValue (const Key *key)`
- `ssize_t keyGetValueSize (const Key *key)`
- `ssize_t keyGetString (const Key *key, char *returnedString, size_t maxSize)`
- `ssize_t keySetString (Key *key, const char *newStringValue)`
- `ssize_t keyGetBinary (const Key *key, void *returnedBinary, size_t maxSize)`
- `ssize_t keySetBinary (Key *key, const void *newBinary, size_t dataSize)`
- `const char * keyComment (const Key *key)`
- `ssize_t keyGetCommentSize (const Key *key)`
- `ssize_t keyGetComment (const Key *key, char *returnedComment, size_t maxSize)`
- `ssize_t keySetComment (Key *key, const char *newComment)`

### 7.8.1 Detailed Description

Methods to do various operations on Key values. A key can contain a value in different format. The most likely situation is, that the value is interpreted as text. Use `keyGetString()` for that. You can save any Unicode Symbols and Elektra will take care that you get the same back, independent of your current environment.

In some situations this idea fails. When you need exactly the same value back without any interpretation of the characters, there is `keySetBinary()`. If you use that, its very likely that your Configuration is not according to the standard. Also for Numbers, Booleans and Date you should use `keyGetString()`. To do so, you might use `strtod()` `strtol()` and then `atol()` or `atof()` to convert back.

To use them:

```
#include <kdb.h>
```

### 7.8.2 Function Documentation

#### 7.8.2.1 `const char* keyComment ( const Key * key )`

Return a pointer to the real internal `key` comment.

This is a much more efficient version of `keyGetComment()` and you should use it if you are responsible enough to not mess up things. You are not allowed to change anything in the memory region the returned pointer points to.

`keyComment()` returns "" when there is no `keyComment`. The reason is

```
key=keyNew(0);
keySetComment(key, "");
```

```
keyComment(key); // you would expect "" here
keyDel(key);
```

See [keySetComment\(\)](#) for more information on comments.

#### Note

Note that the Key structure keeps its own size field that is calculated by library internal calls, so to avoid inconsistencies, you must never use the pointer returned by [keyComment\(\)](#) method to set a new value. Use [keySetComment\(\)](#) instead.

#### Parameters

*key* the key object to work with

#### Returns

a pointer to the internal managed comment  
"" when there is no comment  
0 on NULL pointer

#### See also

[keyGetCommentSize\(\)](#) for size and [keyGetComment\(\)](#) as alternative

#### 7.8.2.2 ssize\_t keyGetBinary ( const Key \* *key*, void \* *returnedBinary*, size\_t *maxSize* )

Get the value of a key as a binary.

If the type is not binary -1 will be returned.

When the binary data is empty (this is not the same as "") 0 will be returned and the returnedBinary will not be changed.

For string values see [keyGetString\(\)](#) and [keyIsString\(\)](#).

When the returnedBinary is too small to hold the data (its maximum size is given by maxSize), the returnedBinary will not be changed and -1 is returned.

#### Example:

```
Key *key = keyNew ("user/keyname", KEY_TYPE, KEY_TYPE_BINARY, KEY_END);
char buffer[300];

if (keyGetBinary(key,buffer,sizeof(buffer)) == -1)
{
    // handle error
}
```

#### Parameters

*key* the object to gather the value from

*returnedBinary* pre-allocated memory to store a copy of the key value

*maxSize* number of bytes of pre-allocated memory in `returnedBinary`

### Returns

the number of bytes actually copied to `returnedBinary`  
0 if the binary is empty  
-1 on NULL pointers  
-1 when `maxSize` is 0, too small to hold the value or larger than `SSIZE_MAX`  
-1 on typing error when the key is not binary

### See also

[keyValue\(\)](#), [keyGetValueSize\(\)](#), [keySetBinary\(\)](#)  
[keyGetString\(\)](#) and [keySetString\(\)](#) as preferred alternative to binary  
[keyIsBinary\(\)](#) to see how to check for binary type

#### 7.8.2.3 `ssize_t keyGetComment ( const Key * key, char * returnedComment, size_t maxSize )`

Get the key comment.

### 7.8.3 Comments

A Key comment is description for humans what this key is for. It may be a textual explanation of valid values, when and why a user or administrator changed the key or any other text that helps the user or administrator related to that key.

Don't depend on a comment in your program. A user is always allowed to remove or change it in any way he wants to. But you are allowed or even encouraged to always show the content of the comment to the user and allow him to change it.

### Parameters

*key* the key object to work with  
*returnedComment* pre-allocated memory to copy the comments to  
*maxSize* number of bytes that will fit `returnedComment`

### Returns

the number of bytes actually copied to `returnedString`, including final NULL  
1 if the string is empty  
-1 on NULL pointer  
-1 if `maxSize` is 0, not enough to store the comment or when larger then `SSIZE_MAX`

### See also

[keyGetCommentSize\(\)](#), [keySetComment\(\)](#)

### 7.8.3.1 `ssize_t keyGetCommentSize ( const Key * key )`

Calculates number of bytes needed to store a key comment, including final NULL.

Use this method to know to size for allocated memory to retrieve a key comment.

See [keySetComment\(\)](#) for more information on comments.

For an empty key name you need one byte to store the ending NULL. For that reason 1 is returned.

```
char *buffer;
buffer = malloc (keyGetCommentSize (key));
// use this buffer to store the comment
// pass keyGetCommentSize (key) for maxSize
```

#### Parameters

*key* the key object to work with

#### Returns

number of bytes needed  
1 if there is no comment  
-1 on NULL pointer

#### See also

[keyGetComment\(\)](#), [keySetComment\(\)](#)

### 7.8.3.2 `ssize_t keyGetString ( const Key * key, char * returnedString, size_t maxSize )`

Get the value of a key as a string.

When there is no value inside the string, 1 will be returned and the returnedString will be empty "" to avoid programming errors that old strings are shown to the user.

For binary values see [keyGetBinary\(\)](#) and [keyIsBinary\(\)](#).

#### Example:

```
Key *key = keyNew ("user/keyname", KEY_END);
char buffer[300];

if (keyGetString(key,buffer,sizeof(buffer)) == -1)
{
    // handle error
} else {
    printf ("buffer: %s\n", buffer);
}
```

#### Parameters

*key* the object to gather the value from

*returnedString* pre-allocated memory to store a copy of the key value  
*maxSize* number of bytes of allocated memory in *returnedString*

### Returns

the number of bytes actually copied to *returnedString*, including final NULL  
 1 if the string is empty  
 -1 on NULL pointer  
 -1 on type mismatch  
*maxSize* is 0, too small for string or is larger than SSIZE\_MAX

### See also

[keyValue\(\)](#), [keyGetValueSize\(\)](#), [keySetString\(\)](#)  
[keyGetBinary\(\)](#) for working with binary data

#### 7.8.3.3 ssize\_t keyGetValueSize ( const Key \* key )

Returns the number of bytes needed to store the key value, including the NULL terminator.

It returns the correct size, independent of the Key Type. If it is a binary there might be '\0' values in it.

For an empty string you need one byte to store the ending NULL. For that reason 1 is returned. This is not true for binary data, so there might be returned 0 too.

A binary key has no '\0' termination. String types have it, so to there length will be added 1 to have enough space to store it.

This method can be used with malloc() before [keyGetString\(\)](#) or [keyGetBinary\(\)](#) is called.

```
char *buffer;
buffer = malloc (keyGetValueSize (key));
// use this buffer to store the value (binary or string)
// pass keyGetValueSize (key) for maxSize
```

### Parameters

*key* the key object to work with

### Returns

the number of bytes needed to store the key value  
 1 when there is no data and type is not binary  
 0 when there is no data and type is binary  
 -1 on null pointer

### See also

[keyGetString\(\)](#), [keyGetBinary\(\)](#), [keyValue\(\)](#)

### 7.8.3.4 `ssize_t keySetBinary ( Key * key, const void * newBinary, size_t dataSize )`

Set the value of a key as a binary.

A private copy of `newBinary` will be allocated and saved inside `key`, so the parameter can be deallocated after the call.

Binary values might be encoded in another way than string values depending on the plugin.

Consider using a string key instead.

When `newBinary` is a NULL pointer the binary will be freed and 0 will be returned.

#### Note

When the key is already binary the meta data won't be changed.

#### Parameters

*key* the object on which to set the value

*newBinary* is a pointer to any binary data or NULL to free the previous set data

*dataSize* number of bytes to copy from `newBinary`

#### Returns

the number of bytes actually copied to internal struct storage

0 when the internal binary was freed

-1 on NULL pointer

-1 when `dataSize` is 0 (but `newBinary` not NULL) or larger than `SSIZE_MAX`

#### See also

[keyGetBinary\(\)](#)

[keyIsBinary\(\)](#) to check if the type is binary

[keyGetString\(\)](#) and [keySetString\(\)](#) as preferred alternative to binary

### 7.8.3.5 `ssize_t keySetComment ( Key * key, const char * newComment )`

Set a comment for a key.

A key comment is like a configuration file comment. See [keySetComment\(\)](#) for more information.

#### Parameters

*key* the key object to work with

*newComment* the comment, that can be freed after this call.

#### Returns

the number of bytes actually saved including final NULL

0 when the comment was freed (`newComment` NULL or empty string)

-1 on NULL pointer or memory problems

See also

[keyGetComment\(\)](#)

#### 7.8.3.6 `ssize_t keySetString ( Key * key, const char * newStringValue )`

Set the value for `key` as `newStringValue`.

The function will allocate and save a private copy of `newStringValue`, so the parameter can be freed after the call.

String values will be saved in backend storage, when `kdbSetKey()` will be called, in UTF-8 universal encoding, regardless of the program's current encoding, when compiled with `--enable-iconv`.

The type will be set to `KEY_TYPE_STRING`. When the type of the key is already a string type it won't be changed.

#### Parameters

*key* the key to set the string value

*newStringValue* NULL-terminated text string to be set as `key`'s value

#### Returns

the number of bytes actually saved in private struct including final NULL  
-1 on NULL pointer

See also

[keyGetString\(\)](#), [keyValue\(\)](#)

#### 7.8.3.7 `const void* keyValue ( const Key * key )`

Return a pointer to the real internal `key` value.

This is a much more efficient version of [keyGetString\(\)](#) [keyGetBinary\(\)](#), and you should use it if you are responsible enough to not mess up things. You are not allowed to modify anything in the returned string. If you need a copy of the Value, consider to use [keyGetString\(\)](#) or [keyGetBinary\(\)](#) instead.

### 7.8.4 String Handling

If `key` is string ([keyIsString\(\)](#)), you may cast the returned as a `"char *"` because you'll get a NULL terminated regular string.

[keyValue\(\)](#) returns "" in string mode when there is no value. The reason is

```
key=keyNew(0);
keySetString(key, "");
keyValue(key); // you would expect "" here
keyDel(key);
```



### 7.8.5 Binary Data Handling

If the data is binary, the size of the value must be determined by `keyGetValueSize()`, any `strlen()` operations are not suitable to determine the size.

`keyValue()` returns 0 in binary mode when there is no value. The reason is

```
key=keyNew(0);
keySetBinary(key, 0, 0);
keyValue(key); // you would expect 0 here

keySetBinary(key, "", 1);
keyValue(key); // you would expect "" (a pointer to '\0') here

int i=23;
keySetBinary(key, (void*)&i, 4);
(int*)keyValue(key); // you would expect a pointer to (int)23 here
keyDel(key);
```

#### Note

Note that the Key structure keeps its own size field that is calculated by library internal calls, so to avoid inconsistencies, you must never use the pointer returned by `keyValue()` method to set a new value. Use `keySetString()` or `keySetBinary()` instead.

#### Warning

Binary keys will return a NULL pointer when there is no data in contrast to `keyName()`, `keyBaseName()`, `keyOwner()` and `keyComment()`. For string value the behaviour is the same.

#### Example:

```
KDB *handle = kdbOpen();
KeySet *ks=ksNew(0);
Key *current=0;

kdbGetByName(handle, ks, "system/sw/my", KDB_O_SORT|KDB_O_RECURSIVE);

ksRewind(ks);
while(current=ksNext(ks)) {
    size_t size=0;

    if (keyIsBin(current)) {
        size=keyGetValueSize(current);
        printf("Key %s has a value of size %d bytes. Value: <BINARY>\nCom
ment: %s",
                keyName(current),
                size,
                keyComment(current));
    } else {
        size=elektraStrLen((char *)keyValue(current));
        printf("Key %s has a value of size %d bytes. Value: %s\nComment:
%s",
                keyName(current),
                size,
                (char *)keyValue(current),
                keyComment(current));
    }
}
```

```

    }
}

ksDel (ks);
kdbClose (handle);

```

### Parameters

**key** the key object to work with

### Returns

a pointer to internal value  
 "" when there is no data and key is not binary  
 0 where there is no data and key is binary  
 0 on NULL pointer

### See also

[keyGetValueSize\(\)](#), [keyGetString\(\)](#), [keyGetBinary\(\)](#)

## 7.9 Interface for mounting backends

### Functions

- int [elektraMountOpen](#) (KDB \*kdb, KeySet \*config, KeySet \*modules, Key \*errorKey)
- int [elektraMountDefault](#) (KDB \*kdb, KeySet \*modules, Key \*errorKey)
- int [elektraMountModules](#) (KDB \*kdb, KeySet \*modules, Key \*errorKey)
- int [elektraMountVersion](#) (KDB \*kdb, Key \*errorKey)
- int [elektraMountBackend](#) (KDB \*kdb, Backend \*backend, Key \*errorKey)
- Key \* [elektraMountGetMountpoint](#) (KDB \*handle, const Key \*where)
- Backend \* [elektraMountGetBackend](#) (KDB \*handle, const Key \*key)

### 7.9.1 Function Documentation

#### 7.9.1.1 int [elektraMountBackend](#) ( KDB \* *kdb*, Backend \* *backend*, Key \* *errorKey* )

Mounts a backend into the trie.

### Parameters

**kdb** the handle to work with  
**backend** the backend to mount  
**errorKey** the key used to report warnings

### Returns

-1 on failure  
 1 on success

**7.9.1.2 int elektraMountDefault ( KDB \* *kdb*, KeySet \* *modules*, Key \* *errorKey* )**

Reopens the default backend and mounts the default backend if needed.

**Precondition**

Default Backend is closed. `elektraMountOpen` was executed before.

**Parameters**

*kdb* the handle to work with

*modules* the current list of loaded modules

*errorKey* the key used to report warnings

**Returns**

-1 on error

0 on success

**7.9.1.3 Backend\* elektraMountGetBackend ( KDB \* *handle*, const Key \* *key* )**

Lookup a backend handle for a specific key.

The required canonical name is ensured by using a key as parameter, which will transform the key to canonical representation.

Will return handle when no more specific KDB could be found.

If key is 0 or invalid the default backend will be returned.

**Parameters**

*handle* is the data structure, where the mounted directories are saved.

*key* the key, that should be looked up.

**Returns**

the backend handle associated with the key

**7.9.1.4 Key\* elektraMountGetMountpoint ( KDB \* *handle*, const Key \* *where* )**

Lookup a mountpoint in a handle for a specific key.

Will return a key representing the mountpoint or null if there is no appropriate mountpoint e.g. its the root mountpoint.

**Example:**

```

Key * key = keyNew ("system/template");
KDB * handle = kdbOpen();
Key *mountpoint=0;
mountpoint=kdbGetMountpoint(handle, key);

printf("The backend I am using is %s mounted in %s\n",
       keyValue(mountpoint),
       keyName(mountpoint));
kdbClose (handle);
keyDel (key);

```

**Parameters**

*handle* is the data structure, where the mounted directories are saved.

*where* the key, that should be looked up.

**Returns**

the mountpoint associated with the key

#### 7.9.1.5 **int elektraMountModules ( KDB \* *kdb*, KeySet \* *modules*, Key \* *errorKey* )**

Mount all module configurations.

**Parameters**

*kdb* the handle to work with

*modules* the current list of loaded modules

*errorKey* the key used to report warnings

**Return values**

*-1* if not rootkey was found

*0* otherwise

#### 7.9.1.6 **int elektraMountOpen ( KDB \* *kdb*, KeySet \* *config*, KeySet \* *modules*, Key \* *errorKey* )**

Creates a trie from a given configuration.

The config will be deleted within this function.

**Note**

elektraMountDefault is not allowed to be executed before

**Parameters**

*kdb* the handle to work with

*modules* the current list of loaded modules

*config* the configuration which should be used to build up the trie.

*errorKey* the key used to report warnings

### Returns

-1 on failure

0 on success

#### 7.9.1.7 int elektraMountVersion ( KDB \* *kdb*, Key \* *errorKey* )

Mount the version backend

### Parameters

*kdb* the handle to work with

*errorKey* the key used to report warnings

### Return values

0 on success

## 7.10 Split :: Represents splitted keysets

used internally for [kdbSet\(\)](#)

### Functions

- [Split](#) \* [elektraSplitNew](#) (void)
- void [elektraSplitDel](#) ([Split](#) \*keysets)
- void [elektraSplitResize](#) ([Split](#) \*split)
- ssize\_t [elektraSplitAppend](#) ([Split](#) \*split, [Backend](#) \*backend, [Key](#) \*parentKey, int syncbits)
- ssize\_t [elektraSplitSearchBackend](#) ([Split](#) \*split, [Backend](#) \*backend, [Key](#) \*parent)
- int [elektraSplitSearchRoot](#) ([Split](#) \*split, [Key](#) \*parentKey)
- int [elektraSplitBuildup](#) ([Split](#) \*split, [KDB](#) \*kdb, [Key](#) \*parentKey)
- int [elektraSplitDivide](#) ([Split](#) \*split, [KDB](#) \*handle, [KeySet](#) \*ks)
- int [elektraSplitAppoint](#) ([Split](#) \*split, [KDB](#) \*handle, [KeySet](#) \*ks)
- int [elektraSplitGet](#) ([Split](#) \*split, [KDB](#) \*handle)
- int [elektraSplitMerge](#) ([Split](#) \*split, [KeySet](#) \*dest)
- int [elektraSplitSync](#) ([Split](#) \*split)
- int [elektraSplitPrepare](#) ([Split](#) \*split)

### 7.10.1 Detailed Description

used internally for [kdbSet\(\)](#) Splits up a keyset into multiple keysets where each of them will be passed to the correct [kdbSet\(\)](#).

### 7.10.2 Function Documentation

#### 7.10.2.1 `ssize_t elektraSplitAppend ( Split * split, Backend * backend, Key * parentKey, int syncbits )`

Increases the size of split and appends a new empty keyset.

Initializes the element with the given parameters at size-1 to be used.

Will automatically resize split if needed.

#### Parameters

*split* the split object to work with

*backend* the backend which should be appended

*parentKey* the parentKey which should be appended

*syncbits* the initial syncstate which should be appended

#### Return values

-1 if no split is found

#### Returns

the size of split - 1

#### 7.10.2.2 `int elektraSplitAppoint ( Split * split, KDB * handle, KeySet * ks )`

Appoints all keys from ks to yet unsynced splits.

#### Precondition

[elektraSplitBuildup\(\)](#) need to be executed before.

#### Parameters

*split* the split object to work with

*handle* to determine to which backend a key belongs

*ks* the keyset to appoint to split

#### Returns

1 on success

-1 if no backend was found for a key

**7.10.2.3 int elektraSplitBuildup ( Split \* *split*, KDB \* *kdb*, Key \* *parentKey* )**

Walks through the trie and adds all backends below *parentKey*.

Sets syncbits to 2 if it is a default or root backend (which needs splitting).

**Precondition**

*split* needs to be empty, directly after creation with [elektraSplitNew\(\)](#).  
there needs to be a valid defaultBackend but its ok not to have a trie inside KDB.  
*parentKey* must be a valid key! (could be implemented more generally, but that would require splitting up of keysets of the same backend)

**Parameters**

*split* the split object to work with

*kdb* the handle to get information about backends

*parentKey* the information below which key the backends are from interest

**Returns**

always 1

**7.10.2.4 void elektraSplitDel ( Split \* *keysets* )**

Delete a split object.

Will free all allocated resources of a splitted keyset.

**Parameters**

*keysets* the split object to work with

**7.10.2.5 int elektraSplitDivide ( Split \* *split*, KDB \* *handle*, KeySet \* *ks* )**

Splits up the keysets and search for a sync bit.

It does not check if there were removed keys, see [elektraSplitRemove\(\)](#) for the next step.

It does not create new backends, this has to be done by buildup before.

**Precondition**

[elektraSplitBuildup\(\)](#) need to be executed before.

**Parameters**

*split* the split object to work with

*handle* to get information where the individual keys belong

*ks* the keyset to divide

### Returns

0 if there were no sync bits  
1 if there were sync bits  
-1 if no backend was found for a key

#### 7.10.2.6 int elektraSplitGet ( Split \* *split*, KDB \* *handle* )

Does some work after getting of backends is finished.

### Precondition

[elektraSplitAppoint\(\)](#) needs to be executed before.

- check if keys are in correct backend
- remove syncbits
- update usersize and systemsize

### Parameters

*split* the split object to work with  
*handle* the handle to preprocess the keys

### Returns

1 on success  
-1 if no backend was found for a key

#### 7.10.2.7 int elektraSplitMerge ( Split \* *split*, KeySet \* *dest* )

Merges together all parts of split into dest.

### Parameters

*split* the split object to work with  
*dest* the destination keyset where all keysets are appended.

### Returns

1 on success



**7.10.2.8 Split\* elektraSplitNew ( void )**

Allocates a new split object.

Initially the size is APPROXIMATE\_NR\_OF\_BACKENDS.

**Returns**

a fresh allocated split object

**See also**

[elektraSplitDel\(\)](#)

**7.10.2.9 int elektraSplitPrepare ( Split \* *split* )**

Prepares for [kdbSet\(\)](#) mainloop afterwards.

All splits which do not need sync are removed and a deep copy of the remaining keysets is done.

**Parameters**

*split* the split object to work with

**Return values**

0 on success

**7.10.2.10 void elektraSplitResize ( Split \* *split* )**

Doubles the size of how many parts of keysets can be appended.

**Parameters**

*split* the split object to work with

**7.10.2.11 ssize\_t elektraSplitSearchBackend ( Split \* *split*, Backend \* *backend*, Key \* *parent* )**

Determines if the backend is already inserted or not.

**Warning**

If no parent Key is given, the default/root backends won't be searched.

**Parameters**

*split* the split object to work with

*backend* the backend to search for

*parent* the key to check for domains in default/root backends.

#### Returns

pos of backend if it already exist  
-1 if it does not exist

#### 7.10.2.12 int elektraSplitSearchRoot ( Split \* *split*, Key \* *parentKey* )

#### Returns

1 if one of the backends in split has all keys below parentKey  
0 if parentKey == 0 or there are keys below or same than parentKey which do not fit in any of splitted keysets

#### Parameters

*split* the split object to work with

*parentKey* the key which relation is searched for

#### 7.10.2.13 int elektraSplitSync ( Split \* *split* )

Add sync bits everywhere keys were removed.

Only this function can really decide if sync is needed or not.

#### Precondition

split needs to be processed with [elektraSplitDivide\(\)](#) before.

#### Returns

0 if [kdbSet\(\)](#) is not needed  
1 if [kdbSet\(\)](#) is needed

#### Precondition

user/system was splitted before.

#### Parameters

*split* the split object to work with

## 7.11 Internal Datastructure for mountpoints

### Functions

- Backend \* [elektraTrieLookup](#) (Trie \*trie, const Key \*key)
- int [elektraTrieClose](#) (Trie \*trie, Key \*errorKey)

### 7.11.1 Function Documentation

#### 7.11.1.1 `int elektraTrieClose ( Trie * trie, Key * errorKey )`

Closes the trie and all opened backends within.

##### Parameters

*trie* the trie to close

*errorKey* the key used to report warnings

##### Return values

0 on success

#### 7.11.1.2 `Backend* elektraTrieLookup ( Trie * trie, const Key * key )`

Lookups a backend inside the trie.

##### Returns

the backend if found

0 otherwise

##### Parameters

*trie* the trie object to work with

*key* the name of this key will be looked up

## 7.12 Plugins :: Elektra framework for plugins

### Functions

- `Plugin * elektraPluginExport (const char *pluginName,...)`
- `KeySet * elektraPluginGetConfig (Plugin *handle)`
- `void elektraPluginSetData (Plugin *plugin, void *data)`
- `void * elektraPluginGetData (Plugin *plugin)`
- `int elektraDocOpen (Plugin *handle, Key *errorKey)`
- `int elektraDocClose (Plugin *handle, Key *errorKey)`
- `int elektraDocGet (Plugin *handle, KeySet *returned, Key *parentKey)`
- `int elektraDocSet (Plugin *handle, KeySet *returned, Key *parentKey)`
- `Plugin * ELEKTRA_PLUGIN_EXPORT (doc)`

### 7.12.1 Detailed Description

#### 7.12.2 History

##### Since

Since version 0.4.9, Elektra can dynamically load different key storage plugins.  
Since version 0.7.0 Elektra can have multiple backends, mounted at any place in the key database.  
Since version 0.8.0 Elektra backends are composed out of multiple plugins.

#### 7.12.3 Overview

There are different types of plugins for different concerns. The types of plugins handled in this document:

- file storage plugins (also called just storage plugins here)
- filter plugins

See <http://www.libelektra.org/ftp/elektra/thesis.pdf> for an detailed explanation and description of other types of plugins.

A plugin can implement anything related to configuration. There are 5 possible entry points, as described in this document:

- `elektraDocOpen()`
- `elektraDocClose()`
- `elektraDocGet()`
- `elektraDocSet()`
- `elektraDocError()` (not needed by storage or filter plugins)

Depending of the type of plugin you need not to implement all of them.

##### Note

that the Doc within the name is just because the plugin described here is called doc (see `src/plugins/doc/doc.c`). Always replace Doc with the name of the plugin you are going to implement.

See the descriptions below what each of them is supposed to do.

##### 7.12.3.1 Storage Plugins

A filter plugin is a plugin which already receives some keys. It may process or change the keyset. Or it may reject specific keysets which do not meet some criteria.

### 7.12.3.2 Filter Plugins

A storage plugin gets an empty keyset and constructs the information out from a file.

Other persistent storage then a file is not handled within this document because it involves many other issues. For files the resolver plugin already takes care for transactions and rollback.

### 7.12.4 Error and Wanrings

In any case of trouble, use ELEKTRA\_SET\_ERROR and return with -1. You might add warnings with ELEKTRA\_ADD\_WARNING if you think it is appropriate.

### 7.12.5 Function Documentation

#### 7.12.5.1 Plugin\* ELEKTRA\_PLUGIN\_EXPORT ( doc )

All KDB methods implemented by the plugin can have random names, except ELEKTRA\_PLUGIN\_EXPORT. This is the single symbol that will be looked up when loading the plugin, and the first method of the backend implementation that will be called.

You need to use a macro so that both dynamic and static loading of the plugin works.

The first paramter is the name of the plugin. Then every plugin should have: ELEKTRA\_PLUGIN\_OPEN, ELEKTRA\_PLUGIN\_CLOSE, ELEKTRA\_PLUGIN\_GET, ELEKTRA\_PLUGIN\_SET and optionally ELEKTRA\_PLUGIN\_ERROR.

The list is terminated with ELEKTRA\_PLUGIN\_END.

You must use static "char arrays" in a read only segment. Don't allocate storage, it won't be freed.

#### Returns

Plugin

#### See also

[elektraPluginExport\(\)](#)

#### 7.12.5.2 int elektraDocClose ( Plugin \* handle, Key \* errorKey )

Finalize the plugin.

Called prior to unloading the plugin dynamic module. Should ensure that no functions or static/global variables from the module will ever be accessed again.

Make sure to free all memory that your plugin requested at runtime.

After this call, libelektra.so will unload the plugin library, so this is the point to shut-down any affairs with the storage.

**Parameters**

*handle* contains internal information of the plugin

*errorKey* is needed to add warnings using ELEKTRA\_ADD\_WARNING

**Return values**

0 on success

**See also**

[kdbClose\(\)](#)

[elektraPluginGetData\(\)](#), [elektraPluginSetData\(\)](#) and [elektraPluginGetConfig\(\)](#)

### 7.12.5.3 int elektraDocGet ( Plugin \* *handle*, KeySet \* *returned*, Key \* *parentKey* )

Retrieve information from a permanent storage to construct a keyset.

## 7.12.6 Introduction

The [elektraDocGet\(\)](#) function handle everything related to receiving keys.

### 7.12.6.1 Storage Plugins

For storage plugins the filename is written in the value of the parentKey. So the first task of the plugin is to open that file. Then it should parse its content and construct a keyset with all information of that file.

You need to be able to reconstruct the same file with the information of the keyset. So be sure to copy all comments, whitespaces and so on into some metadata of the keys. Otherwise the information is lost after writing the file the next time.

Now lets look at an example how the typical [elektraDocGet\(\)](#) might be implemented. To explain we introduce some pseudo functions which do all the work with the storage (which is of course 90% of the work for a real plugin):

- `parse_key` will parse a key and a value from an open file handle

The typical loop for a storage plugin will be like:

```
int elektraDocGet(Plugin *handle, KeySet *returned, const Key *parentKey)
{
    // contract handling, see below

    FILE *fp = fopen (keyString(parentKey), "r");
    char *key;
    char *value;

    while ((n = parse_key(fp, &key, &value)) >= 1)
    {
```

```

        Key *read = keyNew(0);
        if (keySetName(read, key) == -1)
        {
            fclose (fp);
            keyDel (read);
            ELEKTRA_SET_ERROR(59, parentKey, key);
            return -1;
        }
        keySetString(read, value);

        ksAppendKey (returned, read);
        free (key);
        free (value);
    }

    if (feof(fp) == 0)
    {
        fclose (fp);
        ELEKTRA_SET_ERROR(60, parentKey, "not at the end of file");
        return -1;
    }

    fclose (fp);

    return 1; // success
}

```

### 7.12.6.2 Filter Plugins

For filter plugins the actual task is rather unspecified. You basically can do anything with the keyset. To get roundtrip properties you might want to undo any changes you did in [elektraDocSet\(\)](#).

The pseudo functions (which do the real work) are:

- `do_action()` which processes every key in this filter

```

int elektraDocGet(Plugin *handle, KeySet *returned, Key *parentKey)
{
    // contract handling

    Key *k;
    ksRewind (returned);
    while ((k = ksNext (returned)) != 0)
    {
        do_action(k);
    }

    return 1; // success
}

```

### 7.12.7 Conditions

#### Todo

needs some updates

### Precondition

The caller `kdbGet()` will make sure before you are called that the `parentKey`:

- is a valid key (means that it is a system or user key).
- is below (see `keyIsBelow()`) your mountpoint and that your plugin is responsible for it. and that the returned:
- is a valid keyset.
- has all keys with the flag `KEY_FLAG_SYNC` set.
- contains only valid keys direct below (see `keyIsDirectBelow()`) your `parentKey`. That also means, that the `parentKey` will not be in that keyset.
- is in a sorted order, see `ksSort()`. and that the handle:
  - is a valid KDB for your plugin.
  - that `elektraPluginGetBackendHandle()` contains the same handle for lifetime `kdbOpen()` until `elektraPluginClose()` was called.

The caller `kdbGet()` will make sure that afterwards you were called, whenever the user requested it with the options, that:

- hidden keys they will be thrown away.
- dirs or only dirs `kdbGet()` will remove the other.
- you will be called again recursively with all subdirectories.
- the keyset will be sorted when needed.
- the keys in returned having `KEY_FLAG_SYNC` will be sorted out.

### Invariant

There are no global variables and `elektraPluginSetData()` stores all information. The handle is to be guaranteed to be the same if it is the same plugin.

### Postcondition

The keyset returned has the `parentKey` and all keys direct below (`keyIsDirectBelow()`) with all information from the storage. Make sure to return all keys, all directories and also all hidden keys. If some of them are not wished, the caller `kdbGet()` will drop these keys, see above.

## 7.12.8 Updating

To get all keys out of the storage over and over again can be very inefficient. You might know a more efficient method to know if the key needs update or not, e.g. by stating it or by an external time stamp info. For file storage plugins this is automatically done for you. For other types (e.g. databases) you need to implement your own resolver doing this.

### Note

If any calls you use change `errno`, make sure to restore the old `errno`.



**See also**

[kdbGet\(\)](#) for caller.

**Parameters**

*handle* contains internal information of [opened](#) key database

*returned* contains a keyset where the function need to append the keys got from the storage. There might be also some keys inside it, see conditions. You may use them to support efficient updating of keys, see [Updating](#).

*parentKey* contains the information below which key the keys should be gotten.

**Returns**

1 on success

0 when nothing was to do

-1 on failure, the current key in returned shows the position. use ELEKTRA\_SET\_ERROR of kdberrors.h to define the error code

**7.12.8.1 int elektraDocOpen ( Plugin \* *handle*, Key \* *errorKey* )**

Initialize the plugin.

This is the first method called after dynamically loading this plugin.

This method is responsible for:

- plugin's specific configuration gathering
- all plugin's internal structs initialization
- if unavoidable initial setup of all I/O details such as opening a file, connecting to a database, setup connection to a server, etc.

You may also read the configuration you can get with [elektraPluginGetConfig\(\)](#) and transform it into other structures used by your plugin.

**Note**

The plugin must not have any global variables. If you do Elektra will not be threadsafe. It is not a good assumption that your plugin will be opened only once.

Instead you can use [elektraPluginGetData\(\)](#) and [elektraPluginSetData\(\)](#) to store and get any information related to your plugin.

The correct substitute for global variables will be:

```
struct _GlobalData{ int global; };
typedef struct _GlobalData GlobalData;
int elektraDocOpen(Plugin *handle, Key *errorKey)
{
    GlobalData *data;
    data=malloc(sizeof(GlobalData));
    data->global = 20;
    elektraPluginSetData(handle, data);
}
```

**Note**

Make sure to free everything you allocate here within [elektraDocClose\(\)](#).

**Returns**

0 on success

**Parameters**

*handle* contains internal information of [opened](#) key database

*errorKey* defines an errorKey

**See also**

[kdbOpen\(\)](#) which will call [elektraDocOpen\(\)](#)  
[elektraPluginGetData\(\)](#), [elektraPluginSetData\(\)](#) and [elektraPluginGetConfig\(\)](#)

### 7.12.8.2 `int elektraDocSet ( Plugin * handle, KeySet * returned, Key * parentKey )`

Store a keyset permanently.

This function does everything related to set and remove keys in a plugin. There is only one function for that purpose to make implementation and locking much easier.

The keyset *returned* was filled in with information from the application using *elektra* and the task of this function is to store it in a permanent way so that a subsequent call of *elektraPluginGet()* can rebuild the keyset as it was before. See the live cycle of a comment to understand:

```
void usercode (Key *key)
{
    keySetComment (key, "mycomment"); // the usercode stores a comment for the key
    ksAppendKey(keyset, key); // append the key to the keyset
    kdbSet (handle, keyset, 0, 0);
}

// so now kdbSet is called
int kdbSet(KDB *handle, KeySet *keyset, Key *parentKey, options)
{
    // find appropriate plugin
    elektraPluginSet (handle, keyset, 0); // the keyset with the key will be passed to this function
}

// so now elektraPluginSet(), which is the function described here, is called
elektraPluginSet(KDB *handle, KeySet *keyset, Key *parentKey)
{
    // the task of elektraPluginSet is now to store the comment
    Key *key = ksCurrent (keyset); // get out the key where the user set the comment before
    char *comment = allocate(size);
    keyGetComment (key, comment, size);
    savetodisc (comment);
}
```

Of course not only the comment, but all information of every key in the keyset `returned` need to be stored permanently. So this specification needs to give an exhaustive list of information present in a key.

### Precondition

The keyset `returned` holds all keys which must be saved permanently for this keyset. The keyset is sorted and rewinded. All keys having children must be true for `keyIsDir()`.

The `parentKey` is the key which is the ancestor for all other keys in the keyset. The first key of the keyset `returned` has the same keyname. The `parentKey` is below the mountpoint, see `kdbhGetMountpoint()`.

The caller `kdbSet` will fulfill following parts:

- If the user does not want hidden keys they will be thrown away. All keys in `returned` need to be stored permanently.
- If the user does not want dirs or only dirs `kdbGet()` will remove the other.
- Sorting of the keyset. It is not important in which order the keys are appended. So make sure to set all keys, all directories and also all hidden keys. If some of them are not wished, the caller `kdbSet()` will sort them out.

### Invariant

There are no global variables and `kdbhGetBackendData()` only stores information which can be regenerated any time. The handle is the same when it is the same plugin.

### Postcondition

The information of the keyset `returned` is stored permanently.

Lock your permanent storage in an exclusive way, no access of a concurrent `elektraPluginSet_plugin()` or `kdbGet()` is possible and these methods block until the function has finished. Otherwise declare `kdbcGetnoLock()`.

### See also

`kdbSet()` for caller.

### Parameters

*handle* contains internal information of `opened` key database

*returned* contains a keyset with relevant keys

*parentKey* contains the information where to set the keys

### Returns

When everything works gracefully return the number of keys you set. The cursor position and the keys remaining in the keyset are not important.

Return 0 on success with no changed key in database

Return -1 on failure.

**Note**

If any calls you use change `errno`, make sure to restore the old `errno`.

**Error**

In normal execution cases a positive value will be returned. But in some cases you are not able to set keys and have to return -1. If you declare `kdbcGetnoError()` you are done, but otherwise you have to set the cause of the error. (Will be added with 0.7.1)

You also have to make sure that `ksGetCursor()` shows to the position where the error appeared.

**7.12.8.3 Plugin\* elektraPluginExport ( const char \* *pluginName*, ... )**

This function must be called by a plugin's `elektraPluginSymbol()` to define the plugin's methods that will be exported.

See [ELEKTRA\\_PLUGIN\\_EXPORT\(\)](#) how to use it for plugins.

The order and number of arguments are flexible (as in [keyNew\(\)](#) and [ksNew\(\)](#)) to let `libelektra.so` evolve without breaking its ABI compatibility with plugins. So for each method a plugin must export, there is a flag defined by `plugin_t`. Each flag tells `kdbPluginExport()` which method comes next. A plugin can have no implementation for a few methods that have default inefficient high-level implementations and to use these defaults, simply don't pass anything to `kdbPluginExport()` about them.

**Parameters**

*pluginName* a simple name for this plugin

**Returns**

an object that contains all plugin informations needed by `libelektra.so`

**7.12.8.4 KeySet\* elektraPluginGetConfig ( Plugin \* *handle* )**

Returns the configuration of that plugin.

**Parameters**

*handle* a pointer to the plugin

**Returns**

keyset to the configuration for that plugin

#### 7.12.8.5 void\* elektraPluginGetData ( Plugin \* *plugin* )

Get a pointer to any plugin related data stored before.

##### Parameters

*plugin* a pointer to the plugin

##### Returns

a pointer to the data

#### 7.12.8.6 void elektraPluginSetData ( Plugin \* *plugin*, void \* *data* )

Store a pointer to any plugin related data.

##### Parameters

*plugin* a pointer to the plugin

*data* the pointer to the data

## 7.13 Elektra Modules :: Elektra framework for loading modules

Loading Modules for Elektra.

### Functions

- int [elektraModulesInit](#) (KeySet \*modules, Key \*error)
- elektraPluginFactory [elektraModulesLoad](#) (KeySet \*modules, const char \*name, Key \*error)
- int [elektraModulesClose](#) (KeySet \*modules, Key \*error)

#### 7.13.1 Detailed Description

Loading Modules for Elektra. Unfortunately there is no portable way to load modules, plugins or libraries. So Elektra needed a framework which abstracts the loading of modules. Depending of the operating system the build system chooses different source files which actually implement the loading of a module.

The goals are:

- to have a list of all loaded modules
- writing module loaders should be easy

- handle and report errors well
- avoid loading of modules multiple times (maybe OS can't handle that well)
- hide the OS dependent handle inside a Key (handle is needed to close module afterwards)

## 7.13.2 Function Documentation

### 7.13.2.1 int elektraModulesClose ( KeySet \* *modules*, Key \* *error* )

Close all modules.

Iterates over all modules and closes each of them.

Finish all affairs with the modules. Delete all keys where the appropriate module could be closed.

If it is not possible to close a module, still try to close all other modules, but report the error with the error key.

#### Parameters

*modules* all modules in this keyset will be closed

*error* a key to append the error information if it is not null

#### Returns

-1 on error  
>=0 otherwise

### 7.13.2.2 int elektraModulesInit ( KeySet \* *modules*, Key \* *error* )

Initialises the module loading system.

Most operating systems will have to do nothing here. Anyway you are required to add the key system/elektra/modules if it was successful.

On error -1 is returned and if error != 0 error information is added to it.

#### Parameters

*modules* an empty keyset

*error* a key to append the error information if it is not null

#### Returns

-1 on error  
>=0 otherwise

### 7.13.2.3 `elektraPluginFactory elektraModulesLoad ( KeySet * modules, const char * name, Key * error )`

Load a library with the given name.

#### Returns

a pointer to the factory which can create the plugin.

Make sure that you first lookup if this module was already loaded. If it was, just return the pointer and you are done.

Otherwise load the module/library given by name. You need to take care that a proper name is used. The name does not have any path, pre- or postfixes.

The next step is to fetch the symbol `elektraPluginFactory`.

If everything was successful append all information to the keyset modules and return the pointer. Take care that you can close the module with that information. All information needs to be stored within `system/elektra/modules/name`. You might want to use an struct and store it there as binary key.

If anything goes wrong dont append anything to modules. Instead report the error to the error key and return with 0.

#### Precondition

the name is not null, empty and has at least one character different to `/`. It is suitable to be used as `keyAddBaseName` without any further error checking.

#### Parameters

***modules*** where to get existing modules from a new module will be added there

***name*** the name for the plugin to load. Note that it does not have any prefixes or postfixes, you need to add them yourself.

***error*** the key to add warnings or report errors

#### Returns

a pointer which can create a Plugin  
0 on error





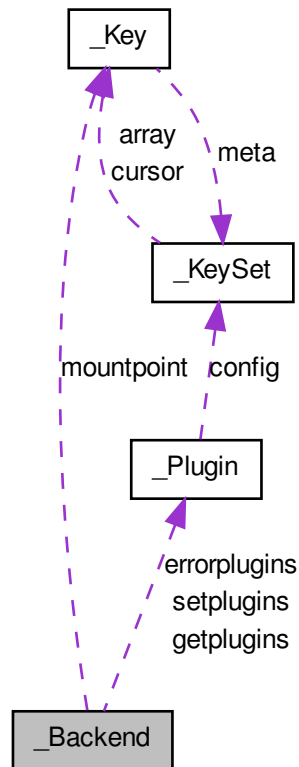
## **Chapter 8**

# **Data Structure Documentation**

### **8.1    \_Backend Struct Reference**

```
#include <kdbprivate.h>
```

Collaboration diagram for `_Backend`:



## Data Fields

- `Key * mountpoint`
- `ssize_t usersize`
- `ssize_t systemsize`
- `size_t refcounter`

### 8.1.1 Detailed Description

Holds all information related to a backend.

Since Elektra 0.8 a Backend consists of many plugins. A backend is responsible for everything related to the process of writing out or reading in configuration.

So this holds a list of set and get plugins.

Backends are put together through the configuration in system/elektra/mountpoints

See kdb mount tool to mount new backends.

To develop a backend you have first to develop plugins and describe through dependencies how they belong together.

## 8.1.2 Field Documentation

### 8.1.2.1 Key\* \_Backend::mountpoint

The mountpoint where the backend resides. The `keyName()` is the point where the backend was mounted. The `keyValue()` is the name of the backend without pre/postfix, e.g. filesystems.

### 8.1.2.2 size\_t \_Backend::refcounter

This refcounter shows how often the backend is used. Not cascading or default backends have 1 in it. More than two is not possible, because a backend can be only mounted in system and user each once.

### 8.1.2.3 ssize\_t \_Backend::systemsize

The size of the systems key from the previous get. Needed to know if a key was removed from a keyset.

### 8.1.2.4 ssize\_t \_Backend::usersize

The size of the users key from the previous get. Needed to know if a key was removed from a keyset.

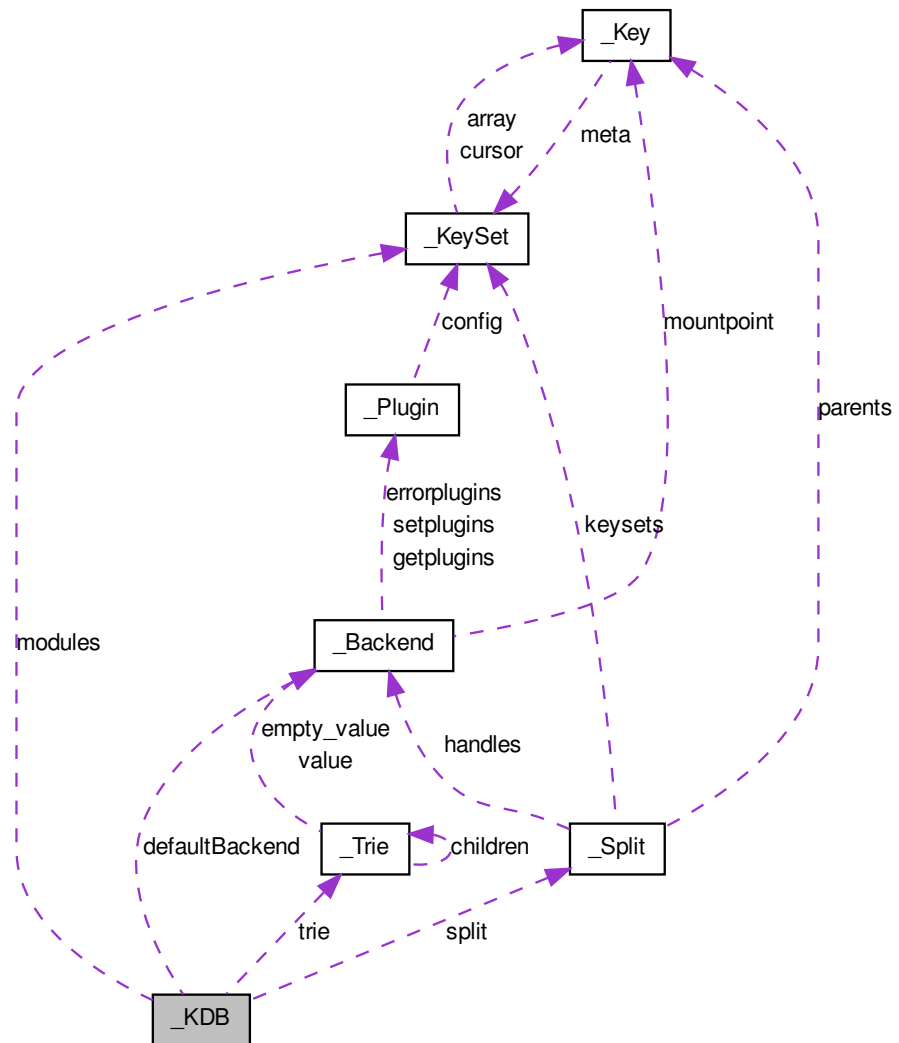
The documentation for this struct was generated from the following file:

- kdbprivate.h

## 8.2 \_KDB Struct Reference

```
#include <kdbprivate.h>
```

Collaboration diagram for \_KDB:



## Data Fields

- `Trie * trie`
- `Split * split`
- `KeySet * modules`
- `Backend * defaultBackend`

### 8.2.1 Detailed Description

The access point to the key database.

The structure which holds all information about loaded backends.

Its internal private attributes should not be accessed directly.

See `kdb mount` tool to mount new backends.

KDB object is defined as:

```
typedef struct _KDB KDB;
```

#### See also

[kdbOpen\(\)](#) and [kdbClose\(\)](#) for external use

### 8.2.2 Field Documentation

#### 8.2.2.1 Backend\* \_KDB::defaultBackend

The default backend as fallback when nothing else is found.

#### 8.2.2.2 KeySet\* \_KDB::modules

A list of all modules loaded at the moment.

#### 8.2.2.3 Split\* \_KDB::split

A list of all mountpoints. It basically has the same information than in the trie, but it is not trivial to convert from one to the other.

#### 8.2.2.4 Trie\* \_KDB::trie

The pointer to the trie holding backends.

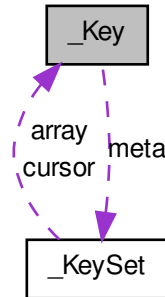
The documentation for this struct was generated from the following file:

- `kdbprivate.h`

## 8.3 **\_Key Struct Reference**

```
#include <kdbprivate.h>
```

Collaboration diagram for `_Key`:



## Data Fields

- union {  
} `data`
- `size_t` `dataSize`
- `char *` `key`
- `size_t` `keySize`
- `keyflag_t` `flags`
- `size_t` `ksReference`
- `KeySet *` `meta`

### 8.3.1 Detailed Description

The private `Key` struct.

Its internal private attributes should not be accessed directly by regular programs. Use the [Key access methods](#) instead. Only a backend writer needs to have access to the private attributes of the `Key` object which is defined as:

```
typedef struct _Key Key;
```

### 8.3.2 Field Documentation

#### 8.3.2.1 union { ... } `_Key::data`

The value, which is a NULL terminated string or binary.

**See also**

[keyString\(\)](#), [keyBinary\(\)](#),  
[keyGetString\(\)](#), [keyGetBinary\(\)](#),  
[keySetString\(\)](#), [keySetBinary\(\)](#)

**8.3.2.2   size\_t \_Key::dataSize**

Size of the value, in bytes, including ending NULL.

**See also**

[keyGetCommentSize\(\)](#), [keySetComment\(\)](#), [keyGetComment\(\)](#)

**8.3.2.3   keyflag\_t \_Key::flags**

Some control and internal flags.

**8.3.2.4   char\* \_Key::key**

The name of the key.

**See also**

[keySetName\(\)](#), [keySetName\(\)](#)

**8.3.2.5   size\_t \_Key::keySize**

Size of the name, in bytes, including ending NULL.

**See also**

[keyGetName\(\)](#), [keyGetNameSize\(\)](#), [keySetName\(\)](#)

**8.3.2.6   size\_t \_Key::ksReference**

In how many keysets the key resists. [keySetName\(\)](#) is only allowed if ksReference is 0.

**See also**

[ksPop\(\)](#), [ksAppendKey\(\)](#), [ksAppend\(\)](#)

### 8.3.2.7 KeySet\* \_Key::meta

All the key's meta information.

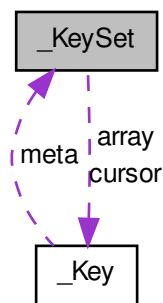
The documentation for this struct was generated from the following file:

- kdbprivate.h

## 8.4 \_KeySet Struct Reference

```
#include <kdbprivate.h>
```

Collaboration diagram for \_KeySet:



### Data Fields

- struct `_Key` \*\* `array`
- `size_t` `size`
- `size_t` `alloc`
- struct `_Key` \* `cursor`
- `size_t` `current`
- `ksflag_t` `flags`

### 8.4.1 Detailed Description

The private KeySet structure.

Its internal private attributes should not be accessed directly by regular programs. Use the [KeySet access methods](#) instead. Only a backend writer needs to have access to the private attributes of the KeySet object which is defined as:



```
typedef struct _KeySet KeySet;
```

## 8.4.2 Field Documentation

### 8.4.2.1 `size_t _KeySet::alloc`

Allocated size of array

### 8.4.2.2 `struct _Key** _KeySet::array`

Array which holds the keys

### 8.4.2.3 `size_t _KeySet::current`

Current position of cursor

### 8.4.2.4 `struct _Key* _KeySet::cursor`

Internal cursor

### 8.4.2.5 `ksflag_t _KeySet::flags`

Some control and internal flags.

### 8.4.2.6 `size_t _KeySet::size`

Number of keys contained in the KeySet

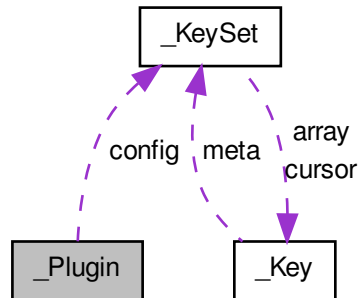
The documentation for this struct was generated from the following file:

- `kdbprivate.h`

## 8.5 \_Plugin Struct Reference

```
#include <kdbprivate.h>
```

Collaboration diagram for `_Plugin`:



## Data Fields

- `KeySet * config`
- `kdbOpenPtr kdbOpen`
- `kdbClosePtr kdbClose`
- `kdbGetPtr kdbGet`
- `kdbSetPtr kdbSet`
- `kdbErrorPtr kdbError`
- `const char * name`
- `size_t refcounter`
- `void * data`

### 8.5.1 Detailed Description

Holds all information related to a plugin.

Since Elektra 0.8 a Backend consists of many plugins.

A plugin should be reusable and only implement a single concern. Plugins which are supplied with Elektra are located below `src/plugins`. It is no problem that plugins are developed external too.

### 8.5.2 Field Documentation

#### 8.5.2.1 `KeySet* _Plugin::config`

This keyset contains configuration for the plugin. Direct below `system/` there is the configuration supplied for the backend. Direct below `user/` there is the configuration

supplied just for the plugin, which should be of course preferred to the backend configuration. The keys inside contain information like `/path` which path should be used to write configuration to or `/host` to which host packets should be sent.

**See also**

[elektraPluginGetConfig\(\)](#)

#### **8.5.2.2 void\* \_Plugin::data**

This handle can be used for a plugin to store any data it wants to.

#### **8.5.2.3 kdbClosePtr \_Plugin::kdbClose**

The pointer to `kdbClose_template()` of the backend.

#### **8.5.2.4 kdbErrorPtr \_Plugin::kdbError**

The pointer to `kdbError_template()` of the backend.

#### **8.5.2.5 kdbGetPtr \_Plugin::kdbGet**

The pointer to `kdbGet_template()` of the backend.

#### **8.5.2.6 kdbOpenPtr \_Plugin::kdbOpen**

The pointer to `kdbOpen_template()` of the backend.

#### **8.5.2.7 kdbSetPtr \_Plugin::kdbSet**

The pointer to `kdbSet_template()` of the backend.

#### **8.5.2.8 const char\* \_Plugin::name**

The name of the module responsible for that plugin.

#### **8.5.2.9 size\_t \_Plugin::refcounter**

This refcounter shows how often the plugin is used. Not shared plugins have 1 in it.

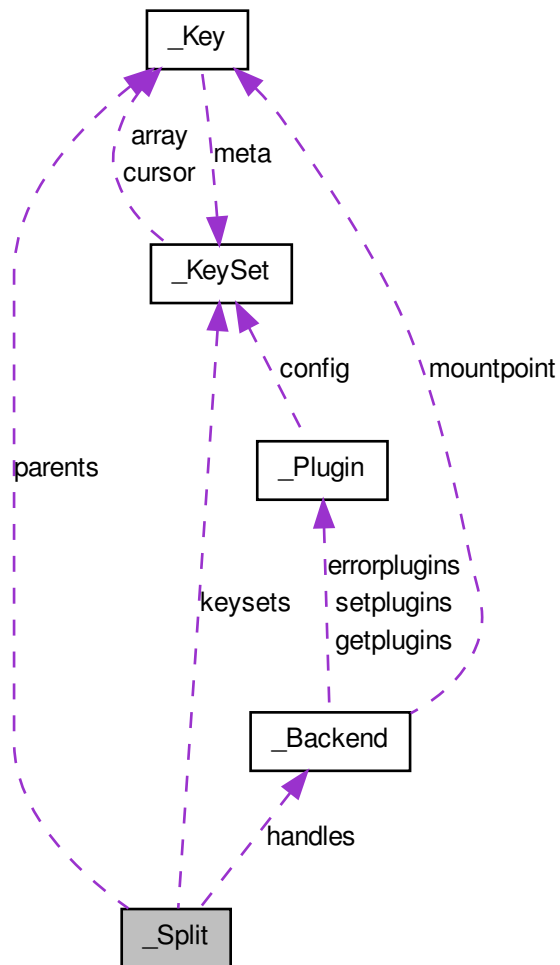
The documentation for this struct was generated from the following file:

- `kdbprivate.h`

## 8.6 \_Split Struct Reference

```
#include <kdbprivate.h>
```

Collaboration diagram for \_Split:



### Data Fields

- `size_t` [size](#)
- `size_t` [alloc](#)

- `KeySet** keysets`
- `Backend** handles`
- `Key** parents`
- `int* syncbits`

### 8.6.1 Detailed Description

The private split structure.

`kdbSet()` splits keysets. This structure contains arrays for various information needed to process the keysets afterwards.

### 8.6.2 Field Documentation

#### 8.6.2.1 `size_t _Split::alloc`

How large the arrays are allocated

#### 8.6.2.2 `Backend** _Split::handles`

The KDB for the keyset

#### 8.6.2.3 `KeySet** _Split::keysets`

The keysets

#### 8.6.2.4 `Key** _Split::parents`

The parentkey for the keyset. Is either the mountpoint of the backend or "user", "system" for the splitted root backends

#### 8.6.2.5 `size_t _Split::size`

Number of keysets

#### 8.6.2.6 `int* _Split::syncbits`

Bits for various options: Bit 0: Is there any key in there which need to be synced? Bit 1: Do we need relative checks? (cascading backend?)

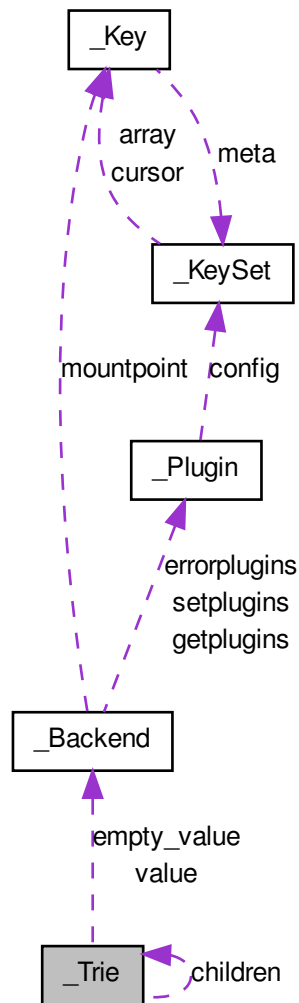
The documentation for this struct was generated from the following file:

- `kdbprivate.h`

## 8.7 \_Trie Struct Reference

```
#include <kdbprivate.h>
```

Collaboration diagram for \_Trie:



### Data Fields

- struct `_Trie` \* `children` [MAX\_UCHAR]

- char \* [text](#) [MAX\_UCHAR]
- size\_t [textlen](#) [MAX\_UCHAR]
- Backend \* [value](#) [MAX\_UCHAR]
- Backend \* [empty\\_value](#)

### 8.7.1 Detailed Description

The private trie structure.

A trie is a data structure which can handle the longest prefix matching very fast. This is exactly what needs to be done when using [kdbGet\(\)](#) and [kdbSet\(\)](#) in a hierarchy where backends are mounted - you need the backend mounted closest to the parentKey.

### 8.7.2 Field Documentation

#### 8.7.2.1 struct \_Trie\* \_Trie::children[MAX\_UCHAR]

The children building up the trie recursively

#### 8.7.2.2 Backend\* \_Trie::empty\_value

Pointer to a backend for the empty string ""

#### 8.7.2.3 char\* \_Trie::text[MAX\_UCHAR]

Text identifying this node

#### 8.7.2.4 size\_t \_Trie::textlen[MAX\_UCHAR]

Length of the text

#### 8.7.2.5 Backend\* \_Trie::value[MAX\_UCHAR]

Pointer to a backend

The documentation for this struct was generated from the following file:

- [kdbprivate.h](#)