The dklibs library set, version 1.21.0

Dipl.-Ing. D. Krause

February 16, 2009

Contents

1 Overview

This package contains a set of libraries.
The main purpose of the libraries is to support application development.
The *dkport* library provides a portability layer and hides system-specific code from the application developer. The *dkc* library contains reusable code for different purposes, i.e.:

- sorted data storage using AVL-trees
- I/O abstraction layer
- application features
  - string tables for internationalization
  - file search
  - logging

The *dknet* library contains code for portable TCP/IP network access.
The *dktrace* library is needed if you use the tracecc preprocessor for debug messages.

2 License

The software in this package is licensed to you under the terms of a BSD style license as follows:

- Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above opyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of the Dirk Krause nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
- This software is provided by the copyright holders and contributors "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed.
  In no event shall the copyright owner or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

3 Usage guidelines

- Do not instantiate types from "dktypes.h" statically in the application code (except the simple data types).
- Use the "..._open/new" functions instead to allocate the variables dynamically.
  These allocation functions are placed in the same modules as the functions accessing these data types so they use the same data type definitions.
- Do not access member variables directly, use the functions provided in the modules instead.

4 Tutorial

4.1 About the tutorial

This tutorial gives an introduction to some of the library modules.
It is restricted to typical usage cases.
It does not cover all the functions contained in the modules, read more in the "Headers and modules" section.
Only minimum error checking is shown in the examples, many of the *if()* statements would normally have an *else* counterpart issuing error messages or warnings.

4.2 Introduction to the dkapp module

4.2.1 The simple Hello-world-program

In this tutorial we will expand the following simple program "hello01.c":

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(int argc, char *argv[])
6 {
7   printf("Hello world!\n");
8   printf("Goodbye world!\n");
9   exit(0);
10  return 0;
11 }
```

4.2.2 Adding application support

To add application support we change the source to “hello02.c”:

```
1
2 #include <dkapp.h>
3
4 #include <stdio.h>
5 #if DK_HAVE_STDLIB_H
6 #include <stdlib.h>
7 #endif
8
9 dk_app_t *app = NULL;
10
11 static int success = 0;
12
13 int main(int argc, char *argv[])
14 {
15   app = dkapp_open(argc, argv);
16   if(app) {
17     printf("Hello world!\n");
18     printf("Goodbye world!\n");
19     success = 1;
20     dkapp_close(app); app = NULL;
21   }
22   success = (success ? 0 : 1);
23   exit(success);
24   return success;
25 }
```

This source must be linked using the libraries `-ldkc -ldkport -lz -lm`. If the dk libraries were compiled without zlib support `-lz` is not needed here.
If the application is started by simply typing

```
1   ./hello02
```

the expected output

```
1   Hello world!
2   Goodbye world!
```

appears.
The command

```
1   ./hello02 --/log/stdout/level=progress
```

already shows some logging.
The “--/...” arguments can be used to override preference settings in the configu-

ration files or registry keys.

Here it is used to set the required message priority for printing to *stdout* down to "progress".

```
Application "hello02" started.
Hello world!
Goodbye world!
Application "hello02" finished.
```

If you type

```
./hello02 --/log/file/keep=debug
```

you will find a file "hello02.3181.log" (the number in the middle can differ) having contents like

```
# 2001/02/14 19:01:31
Application "hello02" started.
# 2001/02/14 19:01:31
Application "hello02" finished.
```

4.2.3 Setting preferences

In your home directory create a subdirectory ".defaults" on U*x or "defaults" on W*.
In this directory we need some files named "hello02", "hello03", "hello04" ....
The contents of the files should be as follows:

```
1  /ui/lang            = en
2  /dir/app            = /home/myname/src/share/Docu/tutorial
3  /log/file/level     = debug
4  /log/file/keep      = error
5  /log/stderr/level   = progress
```

Write the name of the tutorial directory you use instead of "/home/myname/src/share/Docu/tutorial".
The other settings advise the programs to write debug messages and all higher priorized messages to the log file but the logfile is deleted at the processes end unless there was an error message or a higher priorized message.
All progress indication messages and higher priorized messages are also shown on standard error output.
The different priorities are expressed as

- none
  (no messages are issued to the specified destination),
- panic
  (the system is unusable),
- fatal
  (unrecoverable error, program should exit immediately),
- error,
- warning,
- info
  (things that might be of interest),
- progress
  (show the user what we are doing) and
- debug.

4.2.4 Internationalization

Now we want to print messages in the users preferred language.
We enhace our program to “hello03.c” as follows:

```
1  #include <dkapp.h>
2
3  #include <stdio.h>
4
5  #if DK_HAVE_STDLIB_H
6  #include <stdlib.h>
7  #endif
8
9  dk_app_t *app = NULL;
10
11 static int success = 0;
12
13 static char *messages[2];
14
15 static dk_string_finder_t finder[] = {
16   { "hello",   &(messages[0]), "Hello world!" },
17   { "goodbye", &(messages[1]), "Goodbye world!" },
18   { NULL, NULL, NULL }
19 };
20
21 int main(int argc, char *argv[])
22 {
23   app = dkapp_open(argc, argv);
24   if(app) {
25     dkapp_find_multi(app, finder, "h03msg");
26     printf("%s\n", messages[0]);
27     printf("%s\n", messages[1]);
28     success = 1;
29     dkapp_close(app); app = NULL;
30   }
31   success = (success ? 0 : 1);
32   exit(success);
33   return success;
34 }
```

When starting "./hello03" output is as follows:

```
Warning:  String table "h03msg" not found!.
Hello world!
Goodbye world!
```

We are notified that no string table "h03msg" was found.
To create string tables we prepare a file "h03msg.str" as follows:

```
# hello03 string table

"hello"
en =            "Hello, I am an internationalized program."
sp =            "Buenos dias."
de =            "Moin moin."

"goodbye"
en =            "Now it's time to say goodbye."
sp =            "Adios muchachos."
de =            "Und wech."
```

A "#" and the following text until the end of line are comments.
The file consists of a list of entries. Each entry is started by a unique key, i.e. "hello" or "goodbye". The key is followed by a list of language/value pairs.
The language information consists of the language itself and optionally information about region and encoding, so "en_US", "en.UTF-8" and "en_US.UTF-8" are acceptable values too.
In the value the special codes \t, \", \n, \r and \0 can be used. The binary string tables are created by calling

```
stc h03msg.str .
```

This creates subdirectories for the different languages in the current working directory, the directories are named "en", "de" and "sp".
Now start "./hello03" again.
Also test

```
./hello03 --/dir/app=. --/ui/lang=de
./hello03 --/dir/app=. --/ui/lang=sp
```

The "--/dir/app=..." option is only necessary if there is a default preference configured pointing to another directory.

To establish defaults for the preferences you need to find out the current working directory (use “pwd” on *n*x).
Now write a file “$HOME/.defaults/hello03” having a contents like this:

```
/ui/lang        =       en
/ui/lang/env    =       off
/dir/app        =       /home/jim/programming/c/tests
```

For “/ui/lang” choose your preferred language (one of the three in the text string table).
The “/ui/lang/env” setting tells the application to ignore the “LANG” environment variable.
The directory must be replaced by the current directory estimated above (where you run the test programs).

Now lets revisit the source againg.
The array messages is an array of pointers to the localized messages. The array elements must be initialized before they can be used.
This is done by *dkapp_find_multi(app, finder, "h03msg");*. We use the *app* pointer created by *dkapp_open()* before. Binary string tables are searched in files named "h03msg.stt".
The *finder* array contains information about the pointers to set. Each entry consists of the key, the address of the string pointer to set and a default value to be used if no string table or no matching entry is found.
The *finder* array is finished by an element having all components set to NULL.
When using string tables be aware of some traps:

- When adding elements to the *dk_string_finder_t* make sure to increase the size of the string array appropriately.
- Never *printf()*/*fprintf()* the message directly, i.e. by
  printf(messages[0]);
  Instead use *fputs(messages[0],stdout);*
  or *printf("%s",messages[0]);*.
  The strings found in the string table might contain percent signs which are treated as format string beginners.
- Do not forget the finishing element in the *dk_string_finder_t* array.

4.2.5 Logging

We change our program to “hello04.c” as follows:

```
1  #include <dklogc.h>
2  #include <dkapp.h>
3
4  #include <stdio.h>
5  #if DK_HAVE_STDLIB_H
6  #include <stdlib.h>
7  #endif
8
9  dk_app_t *app = NULL;
10
11 static int success = 0;
12
13 static char *messages[7];
14
15 static dk_string_finder_t finder[] = {
16   { "/msg/pan",   &(messages[0]),
17     "Test panic message!"
18   },
19   { "/msg/fat", &(messages[1]),
20     "Test fatal error message!"
21   },
22   { "/msg/err", &(messages[2]),
23     "Test error message."
24   },
25   { "/msg/wrn",  &(messages[3]),
26     "Test warning message."
27   },
28   { "/msg/inf",  &(messages[4]),
29     "Test informational message."
30   },
31   { "/msg/prg", &(messages[5]),
32     "Test progress message."
33   },
34   { "/msg/dbg", &(messages[6]),
35     "Test debug message."
36   },
37   { NULL, NULL, NULL }
38 };
39
40 static char *an_array_of_strings[] = {
```

```
41      "Jim ",
42      "and Joe",
43      " are drinking beer.",
44    };
45
46    int main(int argc, char *argv[])
47    {
48      app = dkapp_open(argc, argv);
49      if(app) {
50        dkapp_find_multi(app, finder, "h04msg");
51        dkapp_log_msg(
52          app, DK_LOG_LEVEL_PANIC,
53          &(messages[0]), 1
54        );
55        dkapp_log_msg(
56          app, DK_LOG_LEVEL_FATAL,
57          &(messages[1]), 1
58        );
59        dkapp_log_msg(
60          app, DK_LOG_LEVEL_ERROR,
61          &(messages[2]), 1
62        );
63        dkapp_log_msg(
64          app, DK_LOG_LEVEL_WARNING,
65          &(messages[3]), 1
66        );
67        dkapp_log_msg(
68          app, DK_LOG_LEVEL_INFO,
69          &(messages[4]), 1
70        );
71        dkapp_log_msg(
72          app, DK_LOG_LEVEL_PROGRESS,
73          &(messages[5]), 1
74        );
75        dkapp_log_msg(
76          app, DK_LOG_LEVEL_DEBUG,
77          &(messages[6]), 1
78        );
79        dkapp_log_msg(
80          app, DK_LOG_LEVEL_INFO,
81          an_array_of_strings, 3
82        );
83        success = 1;
```

```
84        dkapp_close(app); app = NULL;
85      }
86      success = (success ? 0 : 1);
87      exit(success);
88      return success;
89    }
```

the *DK_LOG_LEVEL_...* constants. are in “dklogc.h”
The *dkapp_log_msg()* function is used to issue messages. Each message consists of one ore more strings. The function needs a pointer to an array of strings or a pointer to a pointer to a string and the number of strings to use.
Before you run the program “hello04” you need to run

```
1   stc h04msg.str .
```

Now you can set different values in the “/log/...” preferences in “$HOME/.defaults/hello04” and run

```
1   ./hello04
```

4.2.6 Internationalization simplified

In the *dk_string_finder_t* array in hello04.c we need to take care of the address of string pointers, i.e. by setting a component to "&(messages[5])". We also have to care about the length of the *messages* array.
The *dkapp_init_key_value()* function initializes an array of string pointers using an array of key/value pairs. The key component in each pair contains a string table entry name, the value component contains a default value.
The *DK_KEY_VALUE_ARRAY_SIZE()* macro automatically calculates the array size. When appending new entries in the key/value pair array there is no need to correct the messages array size or to do address calculations.

```
1  #include <dklogc.h>
2  #include <dkapp.h>
3
4  #include <stdio.h>
5
6  #if DK_HAVE_STDLIB_H
7  #include <stdlib.h>
8  #endif
9
10 dk_app_t *app = NULL;
11
12 static int success = 0;
13
14
15 static dk_key_value_t kv[] = {
16   { "/msg/pan",
17     "Test panic message!"
18   },
19   { "/msg/fat",
20     "Test fatal error message!"
21   },
22   { "/msg/err",
23     "Test error message."
24   },
25   { "/msg/wrn",
26     "Test warning message."
27   },
28   { "/msg/inf",
29     "Test informational message."
30   },
31   { "/msg/prg",
32     "Test progress message."
```

```
33      },
34      { "/msg/dbg",
35        "Test debug message."
36      },
37    };
38
39    static char
40    *messages[DK_KEY_VALUE_ARRAY_SIZE(kv)];
41
42    static char *an_array_of_strings[] = {
43      "Jim ",
44      "and Joe",
45      " are drinking beer.",
46    };
47
48    int main(int argc, char *argv[])
49    {
50      size_t kv_s;
51      app = dkapp_open(argc, argv);
52      if(app) {
53        kv_s = DK_KEY_VALUE_ARRAY_SIZE(kv);
54        dkapp_init_key_value(
55          app, kv, kv_s, "h04msg", messages
56        );
57        dkapp_log_msg(
58          app, DK_LOG_LEVEL_PANIC,
59          &(messages[0]), 1
60        );
61        dkapp_log_msg(
62          app, DK_LOG_LEVEL_FATAL,
63          &(messages[1]), 1
64        );
65        dkapp_log_msg(
66          app, DK_LOG_LEVEL_ERROR,
67          &(messages[2]), 1
68        );
69        dkapp_log_msg(
70          app, DK_LOG_LEVEL_WARNING,
71          &(messages[3]), 1
72        );
73        dkapp_log_msg(
74          app, DK_LOG_LEVEL_INFO,
75          &(messages[4]), 1
```

```
76          );
77          dkapp_log_msg(
78            app, DK_LOG_LEVEL_PROGRESS,
79            &(messages[5]), 1
80          );
81          dkapp_log_msg(
82            app, DK_LOG_LEVEL_DEBUG,
83            &(messages[6]), 1
84          );
85          dkapp_log_msg(
86            app, DK_LOG_LEVEL_INFO,
87            an_array_of_strings, 3
88          );
89          success = 1;
90          dkapp_close(app); app = NULL;
91        }
92        success = (success ? 0 : 1);
93        exit(success);
94        return success;
95      }
```

4.2.7 Retrieving command line arguments

File “hello05.c” shows how to retrieve the command line arguments *without* preference overrides:

```
1  #include <dkapp.h>
2
3  #include <stdio.h>
4
5  #if DK_HAVE_STDLIB_H
6  #include <stdlib.h>
7  #endif
8
9  dk_app_t *app = NULL;
10
11 static int success = 0;
12
13 int main(int argc, char *argv[])
14 {
15   int my_argc, i;
16   char **my_argv, **lfdptr;
17   app = dkapp_open(argc, argv);
18   if(app) {
19     success = 1;
20     my_argc = dkapp_get_argc(app);
21     my_argv = dkapp_get_argv(app);
22     lfdptr = my_argv;
23     i = 0;
24     while(i < my_argc) {
25       if(*lfdptr) {
26         printf("%5d \"%s\"\n", i, *lfdptr);
27       }
28       i++; lfdptr++;
29     }
30     dkapp_close(app); app = NULL;
31   }
32   success = (success ? 0 : 1);
33   exit(success);
34   return success;
35 }
```

Run

```
./hello05 a b c --/log/stdout/level=progress \
--/ui/lang=en d e
```

(line too long, wrapped) and you will see the output

```
Application "hello05" started.
    0 "./hello05"
    1 "a"
    2 "b"
    3 "c"
    4 "d"
    5 "e"
Application "hello05" finished.
```

appear.
The pointer returned by *dkapp_get_argv()* can not be used after *dkapp_close()*.

4.3 Memory allocation

In lesson 4.4 on page 22 we will create an in-memory-database for bank accounts. For each bank account we store a customer id and the accounts current value. Before we do so we have to define the data type and to create functions to obtain and release the memory needed.
File “stotest.c” shows these functions:

```
1
2   #include <dksto.h>
3   #include <dkmem.h>
4
5   typedef struct {
6     unsigned long customer_id;
7     double        how_much;
8   } bank_account;
9
10  bank_account *
11  new_bank_account(unsigned long cid)
12  {
13    bank_account *back = NULL;
14    back = dk_new(bank_account,1);
15    if(back) {
16      back->customer_id = cid;
17      back->how_much = 0.0;
18    }
19    return back;
20  }
21
22  void
23  delete_bank_account(bank_account *baptr)
24  {
25    if(baptr) {
26      dk_delete(baptr);
27    }
28  }
```

As you can see we do not call *malloc()*/*free()* directly, instead we use the *dk_new()* and *dk_delete()* macros to do so.
dk_delete() expects the data type as argument and the number of datatypes. It returns a pointer to the new element(s).
The *dk_delete()* macro uses the same pointer to release the memory.
Note: This file does not contain a complete program yet, you can create an object module only.

4.4 Sorting and searching

We can sort the bank accounts by customer id and by the account value.
To distinguish we define constants for the sort criteria:

```
1 #define COMPARE_BY_UID        0
2 #define COMPARE_BY_MONEY      1
3 #define COMPARE_AGAINST_UID   2
```

Now we create a comparison function. The arguments to this function are generic pointers (*void* *) which must be converted to the matching data type before they are used.
The third argument specifies which criteria to use for comparison.
In the *case COMPARE_AGAINST_UID*-branch the second *(void *)* pointer is converted to a *(unsigned long *)* pointer. This is used to find the data belonging to a given account id.

```
static
int
compare_bank_accounts(void *bl, void *br, int what)
{
  int back = 0;
  bank_account *bal, *bar;
  bal = (bank_account *)bl; bar = (bank_account *)br;
  switch(what) {
    case COMPARE_BY_MONEY: {
      if(bal->how_much > bar->how_much) {
        back = 1;
      } else {
        if(bal->how_much < bar->how_much) {
          back = -1;
        }
      }
    } break;
    case COMPARE_AGAINST_UID : {
      uidptr = (unsigned long *)br;
      if(bal->customer_id > (*uidptr)) {
        back = 1;
      } else {
        if(bal->customer_id < (*uidptr)) {
          back = -1;
        }
      }
    } break;
    default : {
      if(bal->customer_id > bar->customer_id) {
        back = 1;
      } else {
        if(bal->customer_id < bar->customer_id) {
          back = -1;
        }
      }
    } break;
  }
  return back;
}
```

In the main function we create two containers (*dk_storage_t*). If this succeeded we create iterators for the containers (*dk_storage_iterator_t*).
If this succeeded too we set the comparison functions for the containers. As you can see data in *st_cid* is sorted by customer id, data in *st_val* is sorted by the accounts value.
When we are done with the iterators and containers we release the memory.

```
1  int main(int argc, char *argv[])
2  {
3    dk_storage_t *st_cid, *st_val;
4    dk_storage_iterator_t *it_cid, *it_val;
5
6    st_cid = dksto_open(0); st_val = dksto_open(0);
7    if(st_cid) { it_cid = dksto_it_open(st_cid); }
8    if(st_val) { it_val = dksto_it_open(st_val); }
9    if(st_cid && st_val && it_cid && it_val) {
10     dksto_set_comp(
11       st_cid, compare_bank_accounts,
12       COMPARE_BY_UID
13     );
14     dksto_set_comp(
15       st_val, compare_bank_accounts,
16       COMPARE_BY_MONEY
17     );
18     /* ... banking takes place here */
19   }
20   if(it_val) {
21     dksto_it_close(it_val); it_val = NULL;
22   }
23   if(it_cid) {
24     dksto_it_close(it_cid); it_cid = NULL;
25   }
26   if(st_val) {
27     dksto_close(st_val); st_val = NULL;
28   }
29   if(st_cid) {
30     dksto_close(st_cid); st_cid = NULL;
31   }
32   exit(0); return 0;
33 }
```

After we finished banking operations we have to clean up memory. Code for this is inserted before we continue in the
"+/- remove all bank account data" section.

```
int main(int argc, char *argv[])
{
  dk_storage_t *st_cid, *st_val;
  dk_storage_iterator_t *it_cid, *it_val;
  int can_continue;
  char inputline[256];
  bank_account *baptr1, *baptr2, *baptr3;

  st_cid = dksto_open(0); st_val = dksto_open(0);
  if(st_cid) { it_cid = dksto_it_open(st_cid); }
  if(st_val) { it_val = dksto_it_open(st_val); }
  if(st_cid && st_val && it_cid && it_val) {
    dksto_set_comp(
      st_cid, compare_bank_accounts,
      COMPARE_BY_UID
    );
    dksto_set_comp(
      st_val, compare_bank_accounts,
      COMPARE_BY_MONEY
    );
    /* ... + banking takes place here */
    /* ... - banking takes place here */
    /* ... + remove all bank account data */
    dksto_it_reset(&it_cid);
    while((baptr1
           =(bank_account *)dksto_it_next(&it_cid))
          !=NULL) {
      delete_bank_account(baptr1);
    }
    /* ... - remove all bank account data */
  }
  if(it_val) { dksto_it_close(it_val); it_val = NULL; }
  if(it_cid) { dksto_it_close(it_cid); it_cid = NULL; }
  if(st_val) { dksto_close(st_val); st_val = NULL; }
  if(st_cid) { dksto_close(st_cid); st_cid = NULL; }
  exit(0); return 0;
}
```

Now we are ready to add the functionality. The program reads standard input and expects the following commands:

```
g <account>
s <account> <value>
p a
p v
```

The *g* command retrieves an account's value and prints it, *s* sets a new value for an account, *p a* prints all accounts values sorted by account id and *p v* print all account values sorted by value.
Some possible input is shown in "STOTEST.IN".
The program code consists of a loop reading input line by line. Each line is parsed to find a command, and options as account id and value.
The following code finds the account data for a given account id *accountnumber*:

```
baptr1 = dksto_it_find_like(
  it_cid,
  (void *)(&accountnumber),
  COMPARE_AGAINST_UID
);
```

We are searching in the container which is sorted by account id.
Each account data is compared against a given account id stored in the variable *accountnumber*. A pointer to that variable is passed to the function.
The search criteria COMPARE_AGAINST_UID tells the comparison function to treat the second *void *)* pointer as a pointer to *unsigned long*.
Note: You can specify another search criteria than the containers sorting criteria but make sure the criteria choice makes sense.
In our example it would not be useful to use the COMPARE_AGAINST_UID search criteria on *it_val* because data is sorted by value in the container *st_val*.
The "+/- create new account" section shows how to create a new bank account and insert it into the two containers:

```
baptr1 = dk_new(bank_account,1);
if(baptr1) {
  baptr1->customer_id = accountnumber;
  baptr1->how_much = value;
  if(dksto_add(st_cid,baptr1)) {
    if(!dksto_add(st_val,baptr1)) {
      dksto_remove(st_cid,baptr1);
      delete_bank_account(baptr1);
    }
  } else {
```

```
11          delete_bank_account(baptr1);
12        }
13    }
```

If the account data can not be added to the containers it is destroyed.
The "+/- update existing account" section shows how to update an existing account:

```
1   dksto_remove(st_val,baptr1);
2   baptr1->how_much = value;
3   if(!dksto_add(st_val,baptr1)) {
4     dksto_remove(st_cid,baptr1);
5     delete_bank_account(baptr1);
6   }
```

The reference to the account data is removed from the *dk_storage_t* sorted by value because the position of the account in the container may change. After updating the accounts value it is inserted again.
The "+/- ... print, sorted..." section shows how to traverse a container:

```
1   dksto_it_reset(it_cid);
2   while((baptr1 = dksto_it_next(it_cid)) != NULL) {
3     printf("%10lu\t%10lg\n", baptr1->customer_id,
4       baptr1->how_much
5     );
6   }
```

After resetting the iterator using *dksto_it_reset()*, *dksto_it_next()* is invoked until it returns NULL.

4.5 Generic I/O

The *dk_stream_t* data type provides an I/O interface so functions do not need to know I/O details. The function simply writes to or reads from the *dk_stream_t* and does not need to care whether it is connected to a file, a port (not yet implemented) or a network connection (also not yet implemented). When writing to file we have the choice whether to compress or not and which compression algorithm to select.

4.5.1 Using the generic I/O

We will write a compression/decompression program named fc. The program will take two arguments, a source file name and a destination file name. Depending on the file name extensions ".gz", ".bz2" or other it treats input and output file as "gzip"- or "bzip2" compressed or uncompressed files. The *copy_streams()* function copies from a source stream *src* to a destination stream *dst*.

```
1  static
2  int
3  copy_streams DK_P2(
4    k_stream_t *, dst, dk_stream_t *, src
5  )
6  {
7    int back;
8    char buffer[MY_BUFFERSIZE];
9    int  can_continue;
10   size_t bufused, bufwritten;
11   can_continue = back = 1;
12   while(can_continue) {
13     bufused =
14     dkstream_read(src,buffer,sizeof(buffer));
15     if(bufused) {
16       bufwritten =
17       dkstream_write(dst,buffer,bufused);
18       if(bufwritten != bufused) {
19         can_continue = 0;
20         back = 0;
21       }
22     } else {
23       can_continue = 0;
24     }
25   }
26   return back;
27 }
```

This function has not to care about the streams' details.

The *main()* function is responsible for setting up the stream:

```
1  int main(int argc, char *argv[])
2  {
3    int exval = 0;
4    int inputtype, outputtype;
5    dk_stream_t *instr, *outstr;
6    char *p1, *p2;
7    if(argc == 3) {
8      inputtype = outputtype = 0;
9      p1 = dksf_get_file_type_dot(argv[1]);
10     p2 = dksf_get_file_type_dot(argv[2]);
11     inputtype = get_type(p1);
12     outputtype = get_type(p2);
13     instr = outstr = NULL;
14     switch(inputtype) {
15       case 1:
16         instr =
17         dkstream_opengz(argv[1], "rb", 0, NULL);
18       break;
19       case 2:
20         instr =
21         dkstream_openbz2(argv[1], "rb", 0, NULL);
22       break;
23       default :
24         instr =
25         dkstream_openfile(argv[1], "rb", 0, NULL);
26       break;
27     }
28     switch(outputtype) {
29       case 1:
30         outstr =
31         dkstream_opengz(argv[2], "wb", 0, NULL);
32       break;
33       case 2:
34         outstr =
35         dkstream_openbz2(argv[2], "wb", 0, NULL);
36       break;
37       default:
38         outstr =
39         dkstream_openfile(argv[2], "wb", 0, NULL);
40       break;
41     }
42     if(instr && outstr) {
```

```
43        exval = copy_streams(outstr, instr);
44      }
45      if(outstr) {
46        dkstream_close(outstr); outstr = NULL;
47      }
48      if(instr)  {
49        dkstream_close(instr);  instr  = NULL;
50      }
51    }
52    exval = (exval ? 0 : 1);
53    exit(exval); return exval;
54  }
```

All it has to do is to select the correct *dkstream_open...* function corresponding to the file name.

The *dkstream_. . . _word()* and *dkstream_. . . _uword()* functions read and write 16-bit-words.
The *dkstream_. . . _dword()* and *dkstream_. . . _udword()* functions are for 32-bit-words.
These functions automatically convert to and from *network byte order*.

The *dkstream_rb_string()* and *dkstream_wb_string()* functions can be used to read and write strings.
The string length (including the trailing null-byte) is written first as 16-bit unsigned word, the string follows.
This allows the the *dkstream_rb_string()* function to read the string length first, allocate memory and read the string.
The string obtained from *dkstream_rb_string()* must be freed by *dk_delete()* when it is not longer needed.

There are two functions *dkstream_gets()* and *dkstream_puts()* similar to the *fgets()* and *fputs()* functions.
If there is no *fgets()*-like function in the *dk_stream_t's* underlying mechanism, characters are read one by one until a line is completed.
This may slow down your application.

4.5.2 Writing stream callback functions

Among others each *dk_stream_t* contains a *(void *)* pointer to the supporting data and a *(dk_stream_fct_t *)* pointer to a function doing the real I/O behind the scenes.
The function takes a pointer to a *dk_stream_api_t* object as argument used for passing arguments and return value.

The *cmd* component of the *dk_stream_api_t* object tells the callback function what to do:

- DK_STREAM_CMD_TEST
  is used to check whether the stream supports a given command.
  The command number to check for is passed in *params.cmd*.
  If the operation is supported *return_value* must be set to *1*, otherwise *0*.

- DK_STREAM_CMD_RDBUF
  indicates a read request. The buffer address and buffer length are passed in *params.buffer* and *params.length*.
  On success (anything was read) *return_value* must be set to *1* by the callback function, *params.used* must contain the number of bytes really read. On error the function must set *return_value* to *0*.

- DK_STREAM_CMD_WRBUF
  indicates a write request. If anything was written the function has to set *return_value* to *1*, *results.used* must contain the number of bytes written.
  On error *return_value* must be set to *0*.

- DK_STREAM_CMD_FINISH
  indicates the closing of the stream. All buffers must be flushed, ressources must be released, files must be closed ...

- DK_STREAM_CMD_FINAL
  indicates the end of the streams operations. Buffers are to be flushed. All resources must be kept usable.

- DK_STREAM_CMD_REWIND
  requests to rewind the stream. On success *return_value* must be set to *1*, otherwise *0*.

- DK_STREAM_CMD_FLUSH
  asks for a buffer flush.

- DK_STREAM_CMD_AT_END
  asks whether we are at the streams end during a read operation. If so we have to set *return_value* to *1*, otherwise *0*.

- DK_STREAM_CMD_GETS
  requests to read a line of test. The buffer pointer and length are passed to the function, *1* and *0* are expected in *return_value* to indicate success and error.

- DK_STREAM_CMD_PUTS
  wants to write a null-terminated string. The string pointer is passed to the function, *return_value* must be set to the return status.

4.5.3 Establishing a callback function

It is recommended to write a . . . *stream_open. . . ()* function. This function should first open the underlaying connection and aquire resources. The next step is to call *dkstream_new()*. If *dkstream_new()* fails the underlaying connection must be closed and the resources need to be released.

4.5.4 Callback example

There is no sample code shown here, look in "dkstream.c" instead.
The *file_stream_function()* is the callback function used for regular files.
The *dkstream_openfile()* function opens a file, *name* and *mode* are the same as used for *fopen()*.
If the file is opened successfully a *dk_stream_t* is allocated and set up for the *(FILE *)* pointer returned by fopen, *file_stream_function* is the callback function.