The dklibs library set, version 1.3.7

Dipl.-Ing. D. Krause

November 26, 2003

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

This software is published under the terms of the
GNU Library General Public License, Version 2.
See the COPYING file for license conditions.

3 Acronyms

Acronym	Meaning
RANF	Really a nice feature.

4 Installation

4.1 Installation on *nix systems

- Unpack the distribution.
- Call

  ```
  ./configure
  make
  make install
  ```

  to configure the package, compile the software and install it.
  The usual options can be specified for `configure`.

4.2 Installation on Windows systems

4.2.1 Using the nt.mak file

A makefile is provided for Visual C++ users on NT workstations, the file is named `nt.mak`.
Building these libraries was tested using Visual C++ 5 on Windows NT 4 workstation SP 6a.
The `dkconfd.h` file contains defaults for this system.
A section

```
#if __BORLANDC__ || __BCPLUSPLUS__
...
#endif
```

is contained too. This section is for Borland C++ 3.1 on DOS.
When using Borland/Imprise compilers on Windows systems you need to figure out how to define the settings correctly.
As a starting point you should replace the contents of the `__BORLANDC__` section by the contents of the "`#if _MSC_VER >= 1100`" section.
Starting with version 1.0.5 one can use the makefile `dll.mak` to create *.DLL files.
Before you use the makefiles you may need to apply some changes:

- `VC`
  must point to your Visual C++'s base directory in which the subdirectories `include`, `lib` and `bin` reside.
- `PROGRAMS`
  must point to the directory where all your software is to be installed. The directory's name depends on the language installed on your system. Typically this is `C:/Programs` or `C:/Programme`.
- `VENDOR`
  is the name of the subdirectory for installation and should need no change.
- `LANG`
  specifies your preferred language and must be set to `en` or `de`.
- `ZL_...`
  enables `zlib` support.
  If `zlib` support is enabled for this library set it must be enabled for all the programs using the libraries too.

- `BZL_...`
  enables `bzlib` support.
  If `bzlib` support is enabled for this library set it must be enabled for all the programs using the libraries too.

From your Visual C++'s `bin` directory run `VCVARS32.BAT` to set the environment variables needed. Other compilers may require another procedure.
To build the libraries type

```
nmake -f nt.mak
```

To install the libraries type

```
nmake -f nt.mak install
```

During installation you will see warnings about string tables which were not found. Don't worry about them, you are just about to build these string tables.
Copy all the `*.exe` files into your W* directory or another directory mentioned in the `PATH` environment variable.
Different W* versions have different options for `copy` and `xcopy` so the makefile needs to be modified when using the home product line. RANF!

4.2.2 Manual installation

- Create a new workspace for a Win32 console application named `dkconfig`.
- In this workspace create a project for another Win32 console application named `tracecc`.
- Create a new project for a Win32 static library named `libdkport`.
- Create a new project for a Win32 static library named `libdkio`.
- Add the source files to the projects, see `Makefile.in` which source to add to which project.
- The `libdkio` library depends on the `libdkport` library.
- Both programs depend on both libraries.
- Correct the header search paths for all the projects.
- Build all the projects.
- Copy `tracecc.exe` and `dkconfig.exe` into a directory contained in the PATH environment variable.
- Copy the `libdkio.lib` and `libdkport.lib` libraries into a special LIB directory.
- Copy all the header files into a special INCLUDE directory.

5 Usage guidelines

5.1 A worst case scenario

The reason for the rewrite was to enable dynamic linking. Although I do not like to use shared libraries - or worse: DLLs - shared libraries are a good choice for extensions to existing software.
If a program uses shared libraries it might happen one installs a newer version of the library but does not update all applications using this library.
In the new library the contents of some data types may have changed (most likely these changes add new members to the data types).
If such datatypes are used statically the application has reserved less space for the variables than the new library functions assume.
If a library function tries to access the "last" components of such a variable it accesses memory not belonging to the variable.
Thus, other data is corrupted or a segmentation violation can happen.

5.2 Conclusions

- 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.

6 Headers and modules

All functions returning `int` values return values `!=0` on success and `0` on error if not specified otherwise.
All functions returning pointers return valid pointers on success and `NULL` on errors if not specified otherwise.

6.1 dk.h

If you use the library set you should always include `dk.h` to have configuration information and data types available.

6.2 dkconfig.h - Configuration

This file is included by `dk.h` and includes other files like `config.h` if available or `dkconfd.h` if there is no `config.h` file available.
Additionaly the file `dkconftr.h` is included. This file transforms configure's `HAVE_...` constants into `DK_HAVE_...` constants.
During installation a new `dkconfig.h` file is created by the `dkconfig` program. This file directly contains `DK_HAVE_...` constants.
The transformation to `DK_HAVE_...` constants is done to avoid collisions with other packages `configure` results.

6.3 dkconfd.h - Default configuration

This file contains some default configurations for Windows PCs. If you use another compiler than MSVC 5.0 you should check whether the constants are defined correctly.

6.4 dkproto.h - Prototypes/Declarations

This file is included automatically by `dk.h` and contains macro definitions to handle ANSI-C's prototypes and K&R-C's function declarations.

6.5 dkerror.h - Error codes

This file contains `int` codes for errors.

- `DK_ERR_NONE`
  No error occured.
- `DK_ERR_SYSERR`
  A system error occured, see `errno` for more information.
- `DK_ERR_NOMEM`
  The system could not allocate memory dynamically.
- `DK_ERR_BUFFER_LENGTH`
  The buffer was too small to keep the result.
- `DK_ERR_MATH_OOR`
  After a mathematical operation the result does not fit into the result data type range.
- `DK_ERR_DIV_ZERO`
  A division by `0` occured.
- `DK_ERR_NO_GETHOSTBYNAME`
  A gethostbyname() function is needed but not available.
- `DK_ERR_NO_SUCH_HOST`
  No information about the specified host were found.
- `DK_ERR_TRY_AGAIN`
  The operation is not possible at this time, but can be executed later.
- `DK_ERR_NO_RECOVERY`
  An unexpected server error occured in gethostbyname().
- `DK_ERR_NO_DNS_RESPONSE`
  The response from the DNS server contained no data.
- `DK_ERR_UNKNOWN_ERROR`
  An unknown error occured, no details are available.
- `DK_ERR_INVALID_ARGS`
  Invalid arguments were passed to a function or system call.
- `DK_ERR_GETHOSTNAME_FAILED`
  The function failed to obtain the current host name.

- `DK_ERR_NOT_NOW`
  The operation is not possible at this time but maybe later.
- `DK_ERR_INVALID_FILEHANDLE`
  An invalid file handle was specified.
- `DK_ERR_CONNECTION_CLOSED_BY_PEER`
  A TCP/IP connection was closed by the peer.
- `DK_ERR_INTERRUPTED`
  An operation was interrupted, i.e. by a signal.
- `DK_ERR_NO_OOB_DATA`
  When asking for out of band (urgent) data no such data was available.
- `DK_ERR_NOT_CONNECTED`
  We need to create a connection first.
- `DK_ERR_TIMED_OUT`
  An operation was timed out.
- `DK_ERR_IO`
  An I/O error occured
- `DK_ERR_RESOURCES`
  The required ressources are not available.
- `DK_ERR_AF_NO_SUPPORT`
  The address family is not supported.
- `DK_ERR_MSG_SIZE`
  A message or buffer was too large.
- `DK_ERR_PIPE`
  A write operation was performed on a pipe which has no reader assigned.
- `DK_ERR_NEED_ADDR`
  An address is needed to perform the operation.
- `DK_ERR_HOST_UNREACHABLE`
  The target host is unreachable.
- `DK_ERR_NET_INTERFACE_DOWN`
  The local network interface is down.

- `DK_ERR_NET_UNREACHABLE`
  The target network is unreachable.
- `DK_ERR_ACCESS`
  Access to ressources needs privileges the user does not have.
- `DK_ERR_PROTO_NOT_SUPPORTED`
  The specified protocol is not supported.
- `DK_ERR_ADDRESS_IN_USE`
  The address is already in use.
- `DK_ERR_ADDRESS_NOT_AVAILABLE`
  The address is no available.
- `DK_ERR_ALREADY_CONNECTED`
  There is already a connection established.
- `DK_ERR_CONNECT_IN_PROGRESS`
  A connect operation is already in progress.
- `DK_ERR_CONNECTION_REFUSED_BY_PEER`
  The attempt to connect was refused be the peer.
- `DK_ERR_BUSY`
  The peer is too busy to accept a connection.
- `DK_ERR_STRING_TOO_LONG`
  A string was too long to handle it.
- `DK_ERR_NO_SUCH_FILE`
  No file matching the given pattern was found.
- `DK_ERR_NOT_UNIQUE`
  Multiple file names match the pattern but a unique file name is needed.
- `DK_ERR_FINISHED`
  Traversing a collection is finished, there are no more elements available.
- `DK_ERR_FUNCTION_UNSUPPORTED`
  The functionality is not supported on this system.

6.6 dktypes.h - Data types

This file containes data type definitions. It is automatically included by `dk.h`.

6.7 dkmem.h - Dynamic memory allocation

This module is repsonsible for memory allocation and deallocation.
Depending on your system you might want to use either the `malloc`/`free` or the `farmalloc`/`farfree` function pair. The following functions are defined in the module:

- `void *dkmem_alloc(size_t elsize, size_t nelem);`
  This function allocates memory for `nelem` elements of a data type of size `elsize`.
- `void dkmem_free(void *p);`
  deallocates memory obtained by `dkmem_alloc`.

Other functions are defined to reset, copy and compare memory regions.

- `void dkmem_res(void *ptr, size_t bytes);`
  resets `bytes` bytes at address `ptr` to 0.
- `void dkmem_cpy(void *d, void *s, size_t n);`
  copies `n` bytes from source address `s` to destination address `d`.
- `int dkmem_cmp(void *s1, void *s2, size_t n);`
  compares 2 buffers of `n` bytes.
  If the contents of the buffers is equal `0` is returned, otherwise a non-`0`-value.

Some macros are defined:

- `#define dk_new(t,s)\`
  `(t *)dkmem_alloc(sizeof(t),((size_t)s))`
  can be used for memory allocation.
- `#define dk_delete(p) dkmem_free((void *)(p))`
  can be used to free memory.
- `DK_MEMRES(ptr,sz)`
  resets memory (`sz` bytes starting at address `ptr`).
- `DK_MEMCPY(d,s,n)`
  copies `n` bytes from address `s` to `d`.
- `DK_MEMCMP(a,b,n)`
  compares buffers of size `n` at addresses `a` and `b`.

Example:

```
char text1[] = { "Test-String" };
char text2[sizeof(text1)]; /* same size */
size_t lgt;
char *ptr;
lgt = strlen(text1);
lgt++;
/* get memory dynamically */
ptr = dk_new(char,lgt);
/* check for success */
if(ptr) {
  /* copy from text1 to ptr */
  DK_MEMCPY(ptr,text1,lgt);
  /* set text2 to 0 */
  DK_MEMRES(text2,lgt);
  /* compare text1 and text2 (result should be != 0) */
  printf("text1, text2: %d\n", DK_MEMCMP(text1,text2,lgt));
  /* compare text1 and ptr (result should be == 0) */
  printf("ptr,   text1: %d\n", DK_MEMCMP(ptr,text1,lgt);
  /* release the memory */
  dk_delete(ptr);
}
```

6.8 dkenc - Encoding

The `dkenc` module can be used for encoding changes. The functions `dkenc_ntohl`, `dkenc_ntohs`, `dkenc_htonl` and `dkenc_htons` have the same functionality as `ntohl`, `ntohs`, `htonl` and `htons`.
The conversion routines are useful not only for networking but for portable file saving too.
Some stupid systems have `ntohl`, `ntohs`, `htonl` and `htons` in a DLL dealing with TCP/IP networking. Special initialization functions must be called before using this DLLs functions. RANF!
On PCs without TCP/IP the DLL is typically not installed. RANF!
On PCs with dialup-connection the initialization functions might create a connection. RANF!

To avoid this the functionality was re-implemented.
Another group of encoding problems is the use of UNICODE characters.
UNICODE uses 32 bits per character, the `dk_unicodechar` data type can be used to store these characters.
If a program needs to deal with a given subset of characters (i.e. ISO-LATIN-8859-1) it would waste a lot of memory when using 32 bits for each character.
Compressed representations for UNICODE strings are available as UTF8 or UTF16-encoding.
The `dkenc_uc2utf8` function can be used to create UTF8 from 32-bit-UNICODE.

```
int
dkenc_uc2utf8(
 dk_unicodechar c, dk_utf8 *u8p, int u8l
);
```

converts one 32-bit-UNICODE character into a number of bytes. These bytes are stored in the buffer `u8p`. The buffer length must be given as `u8l`. The function returns the number of bytes used in the buffer or `0` if the buffer is too small.

The `dkenc_utf82uc` function converts from UTF8 to 32-bit-UNICODE.

```
int
dkenc_utf82uc(
 dk_unicodechar *ucp,dk_utf8 *u8p,int u8l,int *u8u
);
```

reads UTF8-encoded data from the buffer `u8p` with a length `u8l`. It builds one 32-bit-UNICODE character (if possible) and stores it in a variable `ucp` points to. The number of bytes from `u8p` used to build the UNICODE character is stored in a variable `u8u` points to.
The function returns `1` on success and `0` on error.

6.9 dkslsupp - Syslog support

The module contains the function

- `int dkslsupp_get_code(char * str);`

to map a syslog description string like `auth.notice` to a numeric value like `37`.
The string consists of facility and priority separated by a dot.
Table 1 lists the facilities, table 2 on the next page lists the priorities. The function returns the corresponding syslog message code or `0` if no matching code was found.

Table 1: Facility keywords

Keyword	Meaning
`auth`	messages related to user authentication
`authpriv`	
`cron`	messages generated by unattended scheduled jobs
`daemon`	messages generated by daemons and services
`ftp`	messages generated by the FTP service
`kern`	messages generated by the OS kernel
`lpr`	messages generated by the print system
`mail`	messages generated by the mail transport system
`news`	messages generated by the NNTP service
`syslog`	messages generated by the syslog service
`user`	messages generated by programs run from normal users
`uucp`	messages generated by the UUCP service
`local0`	site-specific messages
`local1`	
`local2`	
`local3`	
`local4`	
`local5`	
`local6`	
`local7`	

Table 2: Priority keywords

Keyword	Meaning
`emerg`	The system is unusable.
`alert`	An action must be taken immediately.
`crit`	A critical error occured.
`err`	An error occured.
`warning`	A warning condition occured.
`notice`	Notification about a normal but significant condition.
`info`	Information message.
`debug`	A debug-level message.

6.10 dksfc - Preprocessor definitions for file types and permissions

The header file dksfc.h contains preprocessor definitions for file types and permissions used by the dksf module. The following definitions as described in table 3 are available for file types.

Table 3: File types

Preprocessor definition	Meaning
DK_FT_REG	Regular file
DK_FT_DIR	Directory
DK_FT_FIFO	FIFO
DK_FT_CHR	Character device
DK_FT_BLK	Block device
DK_FT_SOCKET	Socket
DK_FT_OTHER	Other file types
DK_FT_SYMLINK	Symbolic link

The following definitions are available for permissions:

Table 4: Permissions

Preprocessor definition	Meaning
DK_PERM_SUID	SUID-Bit
DK_PERM_SGID	SGID-Bit
DK_PERM_VTX	VTX-Bit (sticky-bit)
DK_PERM_U_READ	read permission for owner
DK_PERM_U_WRITE	write permission for owner
DK_PERM_U_EXECUTE	execution permission for owner
DK_PERM_G_READ	read permission for group
DK_PERM_G_WRITE	write permission for group
DK_PERM_G_EXECUTE	execution permission for group
DK_PERM_O_READ	read permission for others
DK_PERM_O_WRITE	write permission for others
	. . . to be continued

	Continuation
DK_PERM_O_EXECUTE	execution permission for others
DK_PERM_CREATE_DIR	mode suggested for directory creation
DK_PERM_CREATE_FILE	mode suggested for file creation

6.11 dksf - Interface to system functions

The `dksf` module contains data types and functions to access system functions.

6.11.1 Information about files

The `dk_stat_t` data type is used to store information about files.

- `dk_stat_t *dkstat_open(char *filename);`
  retrieves information about the file `filename`. A `dk_stat_t` variable is allocated dynamically and filled with the information.
  On success a pointer to the new variable is returned, otherwise (the file does not exist or there is not enough memory available) `NULL`.
  If a valid pointer is returned the memory must be freed by use of `dkstat_close()` when it is not longer needed.
- `void dkstat_close(dk_stat_t *ptr);`
  frees the memory allocated by `dkstat_open()`.
- `int dkstat_filetype(dk_stat_t *ptr);`
  returns an `int` value for the filetype. The value is as follows:
  - `DK_FT_REG`
    for regular files,
  - `DK_FT_DIR`
    for a directory,
  - `DK_FT_FIFO`
    for FIFOs/Pipes,
  - `DK_FT_CHR`
    for character special devices,
  - `DK_FT_BLK`
    for block special devices,
  - `DK_FT_SOCKET`
    for sockets and
  - `DK_FT_OTHER` for all other filetypes.

  If `filename` is a symbolic link this value is or-combined with `DK_FT_SYMLINK`.

Example:

```
void print_filetype(char *filename)
{
  if(filename) {
    dk_stat_t *info;
    info = dkstat_open(filename);
    if(info) {
      int ft;
      ft = dkstat_filetype(info);
      switch(ft & (~DK_FT_SYMLINK)) {
        case DK_FT_REG: {
          printf("regular file: %s\n", filename);
        } break;
        /* ... */
      }
      dkstat_close(info);
    }
  }
}
```

- `int dkstat_permissions(dk_stat_t *ptr);`
  returns the permissions to the file as or-combination of the following constants:
  - `DK_PERM_U_READ`
    Read permission for the file owner.
  - `DK_PERM_U_WRITE`
    Write permission for the file owner.
  - `DK_PERM_U_EXECUTE`
    Execution permission for the file owner.
  - `DK_PERM_G_READ`
    Read permission for the file owners group.
  - `DK_PERM_G_WRITE`
    Write permission for the file owners group.
  - `DK_PERM_G_EXECUTE`
    Execution permission for the file owners group.
  - `DK_PERM_O_READ`
    Read permission for everybody.
  - `DK_PERM_O_WRITE`
    Write permission for everybody.
  - `DK_PERM_O_EXECUTE`
    Execution permission for everybody.
  - `DK_PERM_SUID`
    The set-user-id-bit is set.
  - `DK_PERM_SGID`
    The set-group-id-bit is set.
  - `DK_PERM_VTX`
    The system should try to keep the executable image in memory.
- `unsigned long dkstat_inode(dk_stat_t *ptr);`
  returns the inode number converted to `unsigned long`.
- `unsigned long dkstat_device(dk_stat_t *ptr);`
  returns the device number converted to `unsigned long`.
- `unsigned long dkstat_rdevice(dk_stat_t *ptr);`
  returns the relative device number converted to `unsigned long`.

- `unsigned long dkstat_nlinks(dk_stat_t *ptr);`
  returns the number of links to the file.
- `dk_long_long_unsigned_t dkstat_size(`
  `dk_stat_t *ptr`
  `);`
  returns the file size converted to `dk_long_long_unsigned_t`.
  Note:
  The `dk_long_long_unsigned_t` is a `long long` `unsigned` on systems which support `long long` and `unsigned long` on systems without `long long` support.
  If the system has large file support but no `long long` support the function may produce wrong result for large files.
  It is recommended to use `dkstat_size_ok()` instead.
- `dk_long_long_unsigned_t dkstat_size_ok(`
  `dk_stat_t *ptr, int *ok`
  `);`
  returns the file size converted to `dk_long_long_unsigned_t`.
  The `ok` argument points to a variable used for error notification. If the function produces a wrong result because the system has large file support but no `long long` data type and the file is too large the variable is set to `DK_ERR_MATH_OOR`. Otherwise the variable is left unchanged.
- `long dkstat_uid(dk_stat_t *ptr);`
  returns the file owners UID converted to `long`.
- `long dkstat_gid(dk_stat_t *ptr);`
  returns the file groups GID converted to `long`.
- `char *dkstat_ctime(dk_stat_t *ptr);`
  returns a pointer to a buffer containing the file creation time converted to a string.
  This buffer belongs to the `dk_stat_t` variable, you must never change nor free this buffer.
- `char *dkstat_atime(dk_stat_t *ptr);`
  returns a pointer to a buffer containing the last file access time converted to a string.
- `char *dkstat_mtime(dk_stat_t *ptr);`
  returns a pointer to a buffer containing the last file modification time converted to a string.

6.11.2 Directory traversal

Information for directory traversal is stored in variables of type `dk_dir_t`.

- `dk_dir_t *dkdir_open(char *name);`
  opens a directory for traversal.
  The function returns a valid pointer on success, `NULL` on error.
  The `dkdir_next()` function needs this pointer to traverse the directory.
  If you are done with the directory the variable must be released by calling `dkdir_close()`.

- `void dkdir_close(dk_dir_t *ptr);`
  closes the directory, releases the ressources used to traverse the directory and deallocates the memory.

- `int dkdir_next(dk_dir_t *ptr);`
  tries to find the next directory entry. If there is yet another entry the function returns `!=0`, otherwise `0`.
  On success you can use the following functions to obtain information about the entry found.

- `char *dkdir_get_fullname(dk_dir_t *ptr);`
  returns a pointer to a buffer containing the entries full name.

- `char *dkdir_get_shortname(dk_dir_t *ptr);`
  returns a pointer to a buffer containing the entries short name.

- ```
  long dkdir_uid(dk_dir_t *ptr);
  long dkdir_gid(dk_dir_t *ptr);
  char *dkdir_ctime(dk_dir_t *ptr);
  char *dkdir_atime(dk_dir_t *ptr);
  char *dkdir_mtime(dk_dir_t *ptr);
  int dkdir_filetype(dk_dir_t *ptr);
  int dkdir_permissions(dk_dir_t *ptr);
  unsigned long dkdir_inode(dk_dir_t *ptr);
  unsigned long dkdir_device(dk_dir_t *ptr);
  unsigned long dkdir_rdevice(dk_dir_t *ptr);
  unsigned long dkdir_nlinks(dk_dir_t *ptr);
  dk_long_long_unsigned_t dkdir_size(dk_dir_t *ptr);
  dk_long_long_unsigned_t dkdir_size_ok(
  dk_dir_t *ptr, int *ok
  );
  ```
  retrieve information about the current entry (see `dkstat_...` above).
  ```

Example:

```
void dir_list(char *dirname)
{
  dk_dir_t *ptr;
  if(dirname) {
    ptr = dkdir_open(dirname);
    if(ptr) {
      while(dkdir_next(ptr)) {
        printf("full name:  %s\n", dkdir_get_fullname(ptr));
        printf("short name: %s\n", dkdir_get_shortname(ptr));
        printf("size:       %lu\n", dkdir_size(ptr));
        /* ... */
      }
      dkdir_close(ptr);
    }
  }
}
```

6.11.3 File name expansion

A real operating systems shell automatically expands wildcards in command line arguments.
Poor operating systems leave this up to the application programmer. RANF!
Newer poor operating systems allow to use whitespaces in file names. A real operating systems shell automatically builds one argument from multiple strings surrounded by quotes. Poor shells don't do that and leave this up to the programmer too. RANF!
The `dk_fne_t` data type can be used to expand filenames.

- ```
  dk_fne_t *dkfne_open(
  char *name, int files, int dirs
  );
  ```
  dynamically allocates a new `dk_fne_t` variable and returns a pointer to that variable.
  The variable is initiliazed to expand the given filename `name`. Specify `files` to `1` if names of regular files are an acceptable file name expansion result, otherwise `0`. Specify `dirs` to `1` if names of directories are an acceptable file name expansion result, otherwise `0`. On success a valid pointer is returned, otherwise `NULL`.
  If you are done expanding the filename, the variable must be deallocated by `dkfne_close()`.

- `void dkfne_close(dk_fne_t *ptr);`
  deallocates a variable created by `dkfne_open()`.

- `int dkfne_next(dk_fne_t *ptr);`
  checks whether a further filename can be constructed, returns `1` for success, `0` for error.
  If this function indicates success you can use the following two functions to obtain the filename.

- `char *dkfne_get_fullname(dk_fne_t *ptr);`
  returns the full name.

- `char *dkfne_get_shortname(dk_fne_t *ptr);`
  returns the short name.
  ```

Example:

```
/* resolve * and ? in pattern */
void list_matching_filenames(char *pattern)
{
 dk_fne_t *fn;
 if(pattern) {
  printf("Searching for regular files\n");
  fn = dkfne_open(pattern, 1, 0);
  if(fn) {
   while(dkfne_next(fn)) {
    printf("file %s\n", dkfne_get_shortname(fn));
    printf("long %s\n", dkfne_get_fullname(fn));
   }
  }
  printf("Searching for directories\n");
  fn = dkfne_open(pattern, 0, 1);
  if(fn) {
   while(dkfne_next(fn)) {
    printf("file %s\n", dkfne_get_shortname(fn));
    printf("long %s\n", dkfne_get_fullname(fn));
   }
  }
 }
}
```

6.11.4 Other functions

Getting the current working directory The function

```
int dksf_getcwd(char *buffer, size_t lgt);
```

can be used to obtain the current working directory. The directory name is written into `buffer`, `lgt` specifies the buffer length in bytes.
The function returns `1` on success, `0` on error (buffer too small...).

Finding the executable file for a command The function

```
int
dksf_get_executable(
 char *buf, size_t len, char *cd, char *pr
);
```

finds the file executed when `pr` is typed on the command line and writes the file name into the buffer specified by `buf` with size `len`. The argument `cd` specifies the current working directory.
The function returns `1` on success, `0` on error.

Finding the file type suffix The function

```
char *dksf_get_file_type_dot(char *name);
```

returns a pointer to a filename's file type suffix.
The pointer points to the dot. On error `NULL` is returned.

Combining two file name compounds The function

```
int
dksf_path_combine(
 char *buf, size_t len, char *p1, char *p2
);
```

concatenates the two file name compounds `p1` and `p2` and writes the result into the buffer specified by `buf` and `len`.
Argument `p1` must be an absolute path name, `p2` can be either an absolute path (in this case p1 is ignored) or a path relative to `p1`.
The function returns `1` on success, `0` on error.

Making directories The function

```
int dksf_mkdir(char *path, int mode);
```

creates a new directory with the name specified in `path`.
The `DK_PERM...-` constants are used in `mode` to specify permissions to the directory. The function returns `1` on success, `0` on error.

Deleting files The function

```
int dksf_remove_file(char *filename);
```

deletes the named file. Either the `unlink()` or `remove()` function is called.

Deleting directories The function

```
int dksf_remove_directory(char *filename);
```

can be used to delete a file or directory. All the directories contents including subdirectories will be deleted.

Changing permissions The function

```
int dksf_chmod(char *path, int mode);
```

changes a file or directories permissions.

Getting the current users UID

```
int dksf_have_getuid(void);
long dksf_getuid(void);
int dksf_have_geteuid(void);
long dksf_geteuid(void);
```

The `dksf_have_getuid()` function checks whether the system knows about the concept of user IDs and has a function to get the current user's ID.
If user IDs are available on the system `dksf_getuid()` returns the current user's UID converted to `long`.
Some systems have mechanisms like `su` to use permissions of other user IDs. In this case the new user ID is also referred to as effective user ID.
The `dksf_have_geteuid()` function checks whether the system supports effective user IDs. If so, `dksf_geteuid()` returns the effective user ID converted to `long`.

Getting the current group ID

```
int dksf_have_getgid(void);
long dksf_getgid(void);
int dksf_have_getegid(void);
long dksf_getegid(void);
```

The functions are similar to the functions above but group IDs are checked instead of user IDs.

Dealing with process IDs

```
int dksf_have_getpid(void);
```

can be used to check whether there is a possibility to obtain a process ID, if so

```
long dksf_getpid(void);
```

returns the process ID converted to `long`.

Parent process ID

```
int dksf_have_getppid(void);
```

can be used to check whether there is a possibility to obtain the parent process'es ID, if so

```
long dksf_getppid(void);
```

returns the parent process'es ID converted to `long`.

Dealing with the process group

```
int dksf_have_getpgrp(void);
```

can be used to check whether there is a possibility to obtain the current process'es process group ID, if so

```
long dksf_getpgrp(void);
```

returns this process group ID converted to `long`.

Another processes process group

```
int dksf_have_getpgid(void);
```

can be used to check whether there is a possibility to obtain another process'es process group, if so

```
long dksf_getpgid(long p);
```

returns this process group ID.

Finding the users login name

```
int dksf_get_uname(char *buffer, size_t sz);
int dksf_get_euname(char *buffer, size_t sz);
```

write the users login name into the buffer or the login name belonging to the current effective user ID. The `sz` parameter is the buffer size in bytes.
On *nix systems the usual `getuid()`/`getpwuid()` functions are used.
If the `GetUserNameA()` function is available on W* systems this function is used first.
If this function is not available or fails we try several places in the registry, i.e. `HKLM/System/CurrentControlSet/control:Current User` or `HKLM/Network/Logon:username`. Different W* versions might use different places. RANF!
If your W* version is newer than W* 98 or W* NT 4.0 there is not yet support for registry lookups in the library. In this cases the environment variable `LOGNAME` or `USERNAME` must be set.

Finding a users home directory

```
int dksf_get_home(char *buffer, size_t sz);
int dksf_get_ehome(char *buffer, size_t sz);
```

retrieve the current users home directory or the home directory belonging to the current effective user ID.
On *nix systems the usual `getuid()`/`getpwuid()` functions are used.
On W* systems there is no such API. Instead there are different places for home directories and different environment variables pointing to the directories depending on the W* version. RANF! We need to inspect multiple environment variables, i.e. `HOME`, `USERPROFILE` and `HOMEDRIVE`/`HOMEPATH`.

Finding the host- and domainname

```
int dksf_get_hostname(char *buffer, size_t sz);
int dksf_get_domainname(char *buffer, size_t sz);
```

retrieve host- and domainname of the current host.
On *nix the usual `gethostname()`/`sysinfo()` functions are used.
On Windows the `GetComputerNameA()` function is used if available to determine the hostname. If the function is unavailable or fails several registry settings are tested, I have found host related information in

- `System`
  `CurrentControlSet`
  `Services`
  `Tcpip`
  `Parameters:Hostname`,
- `System`
  `CurrentControlSet`
  `Control`
  `ComputerName`
  `ActiveComputerName:ComputerName`,
- `System`
  `CurrentControlSet`
  `Control`
  `ComputerName`
  `ComputerName:ComputerName`,
- `System`
  `CurrentControlSet`
  `Services`
  `VxD`
  `MSTCP:HostName` and
- `System`
  `CurrentControlSet`
  `Services`
  `VxD`
  `VNETSUP:ComputerName`

depending on the W* version. For future W* versions I expect the places to differ again. RANF!

Finding a directory for temporary files

```
int dksf_get_tempdir(char *buffer, size_t sz);
```

finds a directory for temporary files.
The TMPDIR, TEMP and TMP environment variables are inspected first.
If this fails a set of directory names typically used is inspected.

Check whether writing to a file should be allowed

```
int dksf_allowed_to_write(
 char *fn, int ignchk, int *rsn
);
```

checks whether it should be allowed to open the file `fn` for write access.
If `fn` is a symbolic link write access should be denied if any of the following conditions is fullfilled:

- `DK_SF_SEC_OWNER`
  The link owner is not the owner of the destination file.
- `DK_SF_SEC_WG`
  The link resides in a group writable directory.
- `DK_SF_SEC_WO`
  The link resides in a world writable directory.

The `ignchk` arguments is an or-combination of `DK_SF_SEC_...` values to skip tests.
If write access should be denied and `rsn` is a valid pointer, the variable is set to the `DK_SF_SEC_...` condition for the test responsible for denying write access.

More secure fopen

```
FILE
*dksf_msfo(char *fn, char *m, int ignchk, int *rsn);
```

tries to open the file `fn` in the mode `m`. The `dksf_allowed_to_write()` function is used for security checks, `ignchk` and `rsn` are passed to this function.
If the `dkapp` module is linked to your program it is recommended to use the `dkapp_fopen()` function (see 6.24 on page 80) to open files. This allows to use command line preferences to skip some of the security checks.

Opening files with security check

```
FILE *dksf_fopen(char *fn, char *mode);
```

calls `dksf_allowed_to_write()` and tries to `fopen()` the file on success. If the `dkapp` module is linked to your program it is recommended to use the `dkapp_fopen()` function (see 6.24 on page 80) to open files. This allows to use command line preferences to skip some of the security checks.

Getting the maximum length of a filename

```
long dksf_get_maxpathlen(void);
```

returns the maximum length for filenames in characters.

Getting the maximum number of open files

```
long dksf_get_maxfiles(void);
```

returns the maximum number of open files.

6.12 dkstr - String handling

The `dkstr` module contains some string handling functions. On some systems `strrchr()` is not available. For case insensitive comparisons some systems use `stricmp()`, others `strcasecmp()`.
This module contains – among others – fallback functions for these functions.

- `int dkstr_casecmp(char *s1, char *s2);`
  compares the strings case insensitive and returns `1` if `s1` comes lexigraphically after `s2`, `-1` if `s1` comes lexigraphically before `s2` and `0` if the strings are equal.
  The function uses `strcasecmp()` or `stricmp()` if available, otherwise it has it's own code.

- `char *dkstr_dup(char *s);`
  creates a new copy of the string, memory is allocated dynamically using the `dkmem_alloc()` function.
  The memory should be freed using `dkmem_free()` if it is not longer needed.

- `char *dkstr_chr(char *s, int c);`
  searches for the first occurence of character `c` in string `s` and returns a pointer to that character if found. Otherwise `NULL` is returned.

- `char *dkstr_rchr(char *s, int c);`
  searches for the last occurence of character `c` in string `s` and returns a pointer to that character if found. Otherwise `NULL` is returned.

- `char *dkstr_start(char *s, char *w);`
  searches for the first non-whitespace in string `s` and returns a pointer to that character (or `NULL` if none is found).
  In `w` a set of whitespace characters can be specified.
  If `NULL` is given here a default set of whitespaces is used.

- `void dkstr_chomp(char *s, char *w);`
  removes trailing whitespaces from the string `s`.

- `char *dkstr_next(char *s, char *w);`
  finishes the first substring (set of non-whitespaces) in string `s` by replacing the first whitespace after the substring by a null-byte and returns a pointer to the start of the next substring.

- `int dkstr_find_multi_part_cmd(`
  `char **cmd, char ***cmdset, int cs`
  `);`
  looks up for the command `cmd` consisting of several words in a table `cmdset` of such commands.
  The index of the command in the table is returned or `-1` if it was not found.
  The `cs` variable controls case-sensitivity of search.

- `int dkstr_is_abbr(`
  `char *line, char *pattern, char spec, int cs`
  `);`
  compares the `line` against the `pattern` and returns `1` for equality or `0` if the texts are different.
  If the `pattern` contains the special character `spec` the text in `line` may be abbreviated at this point.
  The `cs` argument controls whether the comparison is done case-sensitive (`1`) or case-insensitive (`0`).
  When calling `dkstr_is_abbr(line, "Te$st", '$', 1)` we get success if line is "Test", "Tes" or "Te".

- `int dkstr_find_multi_part_abbr(`
  `char **cmd, char ***cmdset, char s, int cs`
  `);`
  behaves like `int dkstr_find_multi_part_cmd(char **cmd, char ***cmdset, int cs);` but uses `dkstr_is_abbr()` for comparisons, `s` marks the place of abbreviation.

Example:

```
#include <stdio.h>
#include <dkstr.h>
void main(void)
{
  char test_line[] = { "  This is a test line.  \n" };
  char *ptr1, *ptr2;
  int i;
  printf(
    "Line at the beginning:        >%s<\n",
    test_line
  );
  ptr1 = dkstr_start(test_line, NULL);
  printf(
    "Without leading whitespaces:  >%s<\n",
    ptr1
  );
  dkstr_chomp(ptr1, NULL);
  printf(
    "Without trailing whitespaces: >%s<\n",
    ptr1
  );
  i = 1;
  while(ptr1) {
    ptr2 = dkstr_next(ptr1, NULL);
    printf(
      "Word %d:                          %s\n",
      i++, ptr1
    );
    ptr1 = ptr2;
  }
}
```

6.13 dktok - Group input characters to tokens

6.13.1 Overview

The dktok module can be used to group input characters to tokens.
To do so we need to create a dk_tokenizer_t structure first. After we are done analyzing input we must release this structure. When creating the tokenizer, we need a buffer where the tokens can be stored, we also specify, which characters are quotes, white spaces, single character tokens, line ends or comments starters. A function to process the tokens is specified and a pointer to store additional information.
The function for token processing is of type

```
int fct(void *d, void *tpv, char *s, int *ec)
```

where *d* is a pointer to the memory for string additional information, *tpv* is a pointer to the tokenizer, *s* is a pointer to the buffer containing the token and *ec* is a pointer to a variable which can be set by the function if an error occurs.
The function must return 1 on success, 0 on errors which require aborting the program.

6.13.2 Functions

The module provides the following functions:

- `dk_tokenizer_t *dktok_new(`
  `size_t b,char **q,char *s,char *w,char *n,char c,`
  `dk_fct_tokenizer *f,void *d`
  `);`
  allocates a new tokenizer structure.
  The *b* parameter is the size of the largest token to process in bytes. A buffer to store tokens is allocated too.
  q is a pointer to an array of strings containing quote characters. Each string contains 2 characters, the starting and the finishing quote. The array must be finished by a `NULL` pointer.
  s is a pointer to a string containing all characters which are a token for theirselves.
  w is a pointer to a string containing all characters which are white spaces.
  c is the character which starts comments spanning until the end of line.
  f is the token handling function which is invoked for every token, *d* is a pointer to additional data.
  The dktok_delete() function must be used to release the memory for the tokenizer and all it's contents.

- `void dktok_start(dk_tokenizer_t * tp);`
  initializes the tokenizer. A program must call this function before character processing is started.
- `int dktok_add(dk_tokenizer_t * tp, char c);`
  adds one character to the tokenizer. As long as the result of this function is > 0 we can continue adding further characters.
- `int dktok_stop(dk_tokenizer_t * tp);`
  must be run after adding the last character.
  If the result of this function is > 0 all input was processed successfully.
- `int dktok_get_error_code(dk_tokenizer_t * tp);`
  returns the error code (which was generated by the user defined token handling funtion).
- `unsigned long dktok_get_lineno(`
  `dk_tokenizer_t * tp`
  `);`
  returns the current line number. This can be used for error messages.
- `void dktok_reset_error_code(dk_tokenizer_t * tp);`
  resets the internal error code.

6.13.3 Example

```
#include <stdio.h>
#include <stdlib.h>
#include <dktok.h>

static char *quotes[] = {
  (char *)"\"\"",
  (char *)"''",
  NULL
};

static char sct[] = { "{}=;" };

static char nl[] = { "\n" };

static char whsp[] = { " \t\r\b" };

static char inbuffer[512];
```

```
int fct(void *d, void *tpv, char *s, int *ec)
{
  int back = 1;
  dk_tokenizer_t *tp;
  if(tpv) {
    tp = (dk_tokenizer_t *)tpv;
    $? "%s", TR_STR(s)
    printf("%10lu \"%s\"\n", dktok_get_lineno(tp), s);
  }
  return back;
}

int main(int argc, char *argv[])
{
  size_t sz;
  int cc; char *ptr;
  dk_tokenizer_t *tp;

  $(trace-init dktok.deb)
  tp = dktok_new(256,quotes,sct,whsp,nl,'#',fct,NULL);
  if(tp) {
    dktok_start(tp);
    cc = 1;
    while(cc) {
      sz = fread((void *)inbuffer,1,sizeof(inbuffer),stdin);
      if(sz > 0) {
        ptr = inbuffer;
while((sz--) && cc) {
  if(!dktok_add(tp, *(ptr++))) cc = 0;
}
      } else {
        cc = 0;
      }
    }
    if(!dktok_get_error_code(tp)) { dktok_stop(tp); }
    dktok_delete(tp);
  } else {
    fprintf(stderr, "Not enough memory!\n");
    fflush(stderr);
  }
```

```
  $(trace-end)
  exit(0); return 0;
}
```

6.14 dksignal - Signal handling

In the UNIX world there are three models of signal handling:

- `signal()`
  This is the simplest model, for many purposes it is dangereous to use because a signal handler must reinstall itself.
  If the signal handler is running and the same signal is delivered before the signal handler reinstalled itself the program is terminated.

- `sigset()`
  This allows you to install permanent signal handlers without the race condition shown above.

- `sigaction()`
  This is the POSIX function set for signal handling.

This module unifies signal handling from the application programmers point of view. An application might look like this:

```
#include <stdio.h>
#include <dksignal.h>
#if DK_HAVE_STDLIB_H
#include <stdlib.h>
#endif
#if DK_HAVE_UNISTD_H
#include <unistd.h>
#endif
dk_signal_ret_t handler(int signo)
{
  dksignal_refresh(signo,handler);
  printf("Signal %d occured\n", signo);
}
void main(void)
{
  switch(dksignal_available()) {
    case 1: printf("signal()\n"); break;
    case 2: printf("sigset()\n"); break;
    case 3: printf("sigaction()\n"); break;
    default:
      printf("No signal handling installed.\n");
    break;
  }
```

```
    old_disp = dksignal_set(SIGINT,handler);
    sleep(10);
    (void)dksignal_set(SIGINT, handler);
  }
```

The `dk_signal_ret_t` is the return type for signal handlers. Possibly this must be set to `int` in `dktypes.h` if there are errors/warnings when compiling the module.
The `dk_signal_fct_t` is the prototype for signal handler.
`dk_signal_disp_t` is a pointer to such a function.
The macro `dksignal_refresh()` reinstalls the handler, if only the `signal()` function is available for signal handling. Otherwise this macro does nothing. The `dksignal_available()` function can be used at runtime to get information about the signal handling available. It returns `0` if no signal handling is available, `1` for the `signal()` function, `2` for the `sigset()` function and `3` for POSIX signal handling.
The `dksignal_set()` function installs a new signal handler and returns the address of the previous signal handler.

6.15 dklog - Log messages (obsoleted)

Note: This module is obsoleted. Logging is done by the `dkapp` module now.

6.15.1 Writing log messages

The `dklog` module allows you to write log messages.
Without special setup log messages are written to diagnostic output (`stderr`).
There are different log levels available for different "heavinesses".

- `DK_LOG_LEVEL_NONE`
  No log messages are printed.
- `DK_LOG_LEVEL_PANIC`
  Condititions requiring a system shutdown immediately.
- `DK_LOG_LEVEL_FATAL`
  Unrecoverable error in application.
- `DK_LOG_LEVEL_ERROR`
  An error occured.
- `DK_LOG_LEVEL_WARNING`
  A warning is to be issued.
- `DK_LOG_LEVEL_INFO`
  Print some information the user should know.
- `DK_LOG_LEVEL_PROGRESS`
  Print progress messages while working.
- `DK_LOG_LEVEL_DEBUG`
  Messages for debugging purposes.
- `DK_LOG_LEVEL_IGNORE`
  Message can be ignored (Please do not use this message level).

There are two functions to write log messages:

- `int dklog_msg(int level, char **strings, int num);`
- `int dklog_msg_ss(int level, char **strings, int num);`

The first function expects the log level as first argument, when a pointer to an array of strings to print and the number of strings.
All the strings in the array are printed as they are, it is your turn to insert all the spaces needed...
The second function uses the string search feature (if configured) and can be used for internationalization. The last argument is again the number of strings to print.
For each string to print the array `strings` must contain three elements:

- a string table name (i.e. "messages"),
- a search key (i.e. "error.no-such-file") and
- a default value to be printed if string search is not configured or no matching entry was found (i.e. "File not found!").

String search is explained later.

6.15.2 Customizing log output

The `dklog` module uses objects of type `dk_log_t` to write messages.
There is always one *active* log object.
If you only want to change which log messages are to be ignored and which are to be issued it is sufficient to apply changes to the current active log object.

```
dk_log_t
*dklog_get(void);

int
dklog_get_minlevel(dk_log_t *ptr);

void
dklog_set_minlevel(dk_log_t *ptr, int minlevel);
```

The function `dklog_get()` returns a pointer to the current active log object.
This pointer can be used to retrieve (`dklog_get_minlevel()`) and set (`dklog_set_minlevel(`
the minimum log level.

If you need a more customized logging (i.e. using the syslog feature or printing to file) you need to create your own log function and log object.
First write a log function like:

```
static void
log_to_file(void *vptr, int l, char **p, int n)
{
  FILE *fipo;
  char **ptr, int i;
  fipo = (FILE *)vptr;
  if(fipo && p && n) {
   fprintf(fipo, "Heaviness: %d   Message: ", l);
   ptr = p;
   for(i = 0; i < n; i++) {
    if(*ptr) {
     fprintf(fipo, *ptr);
    }
    ptr++;
   }
   fprintf(fipo, "\n");
  }
}
```

The `vptr` argument is a pointer to some assisting data. If no assisting data is needed (i.e. when logging to syslog) this pointer is `NULL` and can be ignored.
The second argument is the log level.
The remaining arguments are a pointer to an array of pointers to strings and the length of that array.

Now your application may look like this:

```
FILE *logfile;
dk_log_t *old_log, *new_log;
logfile = fopen("logfile.log", "w");
if(logfile) {
 /* create new log structure */
 new_log =
 dklog_new_for((void *)logfile, log_to_file);
 if(new_log) {
  /* get current log settings */
  old_log = dklog_get();
  /* set minimum level */
  dklog_set_minlevel(
    new_log, DK_LOG_LEVEL_WARNING
  );
  /* use new log behaviour */
  dklog_set(new_log);
  /*
    ... application code possibliy doing logs
        can be placed here ...
  */
  /* restore old log settings */
  dklog_set(old_log);
  /* release log object */
  dklog_delete(new_log);
 } else {
   /* ERROR: not enough memory to create new variable */
 }
 fclose(logfile);
} else { /* ERROR: failed to open logfile */
}
```

6.16 dkss - String search (obsoleted)

Note: This module is obsoleted. String search is done by the `dkapp` module now.

String search can be used for internationalization.
The function

```
char *dkss_find(char *table, char *key, char *def);
```

searches in the string table identified by `table` for an entry matching the named `key`.
If a matching entry is found, that entries value is returned, otherwise the default value `def`.
Before you can search for strings the module must be set up. To do so first create a string search function like:

```
char *
my_string_search(
  void *vptr, char *table, char *key, char *def
)
{
  char *back;
  /* back = ... */
  return back;
}
```

This function has to load the named string table into memory if not yet loaded, search for a matching entry and return that entries value if found or `def` if not.
The pointer `vptr` can be used for assisting data.
The memory containing the value must not be overwritten or released by subsequent calls to the function.
Once you have a function your application may look like this:

```
dk_ss_t *old_ss, *new_ss;
/*
 * retrieve pointer to current
 * active string search object
 */
old_ss = dkss_get_object();
/*
 * create new customized
 * string search object
```

```
 */
new_ss =
dkss_new_for((void *)samepointer, my_string_search);
if(new_ss) {
 /*
  * configure new string search object
  * as active object
  */
 dkss_set_object(new_ss);
 /*
   ... internationalized application code ...
   ... calling dkss_find() ...
 */
 /*
  * activate old string search object,
  * de-activate the new one
  */
 dkss_set_object(old_ss);
 /* release the new string search object */
 dkss_delete(new_ss);
} else {
  /*
   * ERROR: Not enough memory,
   * failed to create object
   */
}
```

By default there is no string search object active, `dkss_find()` returns the default value.

6.17 dkbf - Bit fields

The `dkbf` module contains a data type and the following functions to deal with bit fields:

- `dk_bitfield_t *dkbf_open(size_t bits);`
  creates a new bit field and returns a pointer to the new bit field.
  The argument `bits` specifies the length of the field in bits.
  Once you are done with the bit field you must release it using the `void dkbf_close(dk_bitfield_t *bf);` function.
- `void dkbf_set(`
  `dk_bitfield_t *bf, size_t bit, int val`
  `);`
  sets bit number `bit` to 1 if $val \neq 0$ or to 0 if $val = 0$.
- `int dkbf_get(dk_bitfield_t *bf, size_t bit);`
  retrieves the value of bit number `bit` from the bit field.
- `void dkbf_close(dk_bitfield_t *bf);`
  destroys a bitfield and releases the memory associated with it.

6.18 dkma - Mathematical operations

The dkmath module contains functions for mathematical operations on double, long and unsigned long numbers.
Checks are performed to indicate results out of range and to prevent divisions by 0.
The following functions are available:

- *optype* dkma_*mathop_optypeabbr*
  (*optype* o1, *optype*o2
  );

- *optype* dkma_*mathop_optypeabbr*_ok(
  optype o1, *optype*o2, int *ok
  );

opttypeabbr can be l for opttype long, ul for unsigned long and double for double.
mathop specifies which operation to perform, it can be add, sub, mul or div.
The functions return the operations result.
See dkma.h for a list of all functions. The ..._ok-functions store the check result in the variable ok points to. If there is no error the variable is not modified. If there is an error the variable is set to DK_ERR_MATH_OOR (for result out of range) or DK_ERR_DIV_ZERO (for division by zero).
Functions without _ok store the check result in a static variable in the module. The value of this variable can be retrieved by

```
int dkma_get_error(int reset_variable);
```

The function returns the error flag variables value. If reset_variable is !=0 the variable is reset.
The module also contains functions to convert long and unsigned long to double and vice versa, also with range checking.

An application using the module could look like this:

```
void print_circle_area(double radius)
{
  double area;
  /* reset error condition */
  (void)dkma_get_error(1);
  /* area = radius * radius * M_PI; */
  /* do operation */
  area = dkma_mul_double(
    dkma_mul_double(radius,radius),
    M_PI
  );
  /* check and reset error condition */
  if(dkma_get_error(1)) {
    printf("Radius too large: %lg\n", radius);
  } else {
    printf("Area = %lg\n", area);
  }
}
```

The same function could use it’s own error flag:

```
void print_circle_area(double radius)
{
  double area;
  int result_not_ok;

  /* reset error condition */
  result_ok = DK_ERR_NONE;
  /* area = radius * radius * M_PI; */
  /* do operation */
  area = dkma_mul_double_ok(
    dkma_mul_double_ok(
      radius,radius, &result_not_ok
    ),
    M_PI,
    &result_not_ok
  );
  /* check error condition */
  if(result_not_ok) {
    printf("Radius too large: %lg\n", radius);
  } else {
    printf("Area = %lg\n", area);
    unsigned long ularea;
    ularea = dkma_double_to_ul_ok(
      area, &result_not_ok
    );
    if(result_not_ok) {
      printf("Area too large to express");
      printf(" it as unsigned long.\n");
    } else {
      printf("Area (int) = %lu\n", ularea);
    }
  }
  result_ok = DK_ERR_NONE;
}
```

6.19 dkstream - I/O API

6.19.1 Doing I/O operations

I/O-operations are hidden from the application code, the application only uses `dkstream_...` functions. Real I/O is done by handler functions. Information about a stream is contained int the `dk_stream_t` data type.
The following functions are available in the module:

- *Note:* When linking the `dkapp` module into your program you should use
  - `dkapp_stream_openfile()`,
  - `dkapp_stream_opengz()` and
  - `dkapp_stream_openbz2()`

  (see 6.24 on page 80) instead of `dkstream_open...()`.
  This allows to specify command line preferences to skip some of the security checks.

- ```
  dk_stream_t dkstream_openfile(
  char *n, char *m, int ign, int *rsn
  );
  ```
  opens the file `n` in the given mode `m`, creates a `dk_stream_t` variable and sets this `dk_stream_t` up to use the file for I/O operations.
  A pointer to the new `dk_stream_t` variable is returned.
  The variable must be released using `dkstream_close` after finishing I/O.
  The `ign` parameter can be used to *skip* certain tests. An or-combination of `DK_SF_SEC_...` constants can be specified here, see 6.11.4 on page 38.
  If any of the security checks fails the file is not opened, if `rsn` contains a valid pointer the key number of the failed test is stored in that variable.

- ```
  dk_stream_t dkstream_opengz(
  char *n, char *m, int ign, int *rsn
  );
  ```
  opens the file as `gzip`-file in the given mode, allocates memory for a `dk_stream_t` variable and sets this `dk_stream_t` up to use the file for I/O operations.
  A pointer to the new `dk_stream_t` variable is returned.
  The variable must be released using `dkstream_close` after finishing I/O.

- `dk_stream_t dkstream_openbz2(`
  `char *n, char *m, int ign, int *rsn`
  `);`
  opens the file as `bzip2`-file in the given mode, allocates memory for a `dk_stream_t` variable and sets this `dk_stream_t` up to use the file for I/O operations.
  A pointer to the new `dk_stream_t` variable is returned.
  The variable must be released using `dkstream_close` after finishing I/O.
- `void dkstream_close(dk_stream_t *st);`
  releases the resources connected to the `dk_stream_t` (i.e. closes files…) and releases the `dk_stream_t` itself.
- `size_t dkstream_write(`
  `dk_steam_t *st, char *b, size_t l`
  `);`
  writes `l` bytes starting from buffer address `b` to the stream.
  The function returns the number of bytes successfully written.
- `size_t dkstream_read(`
  `dk_stream_t *st, char *b, size_t l`
  `);`
  reads `l` bytes from the specified stream into the buffer starting at address `b`.
  The number of bytes read is returned.
- `char *dkstream_gets(`
  `dk_stream_t *st, char *b, size_t l`
  `);`
  reads a line from the specified stream into the buffer `b` of length `l`.
  Read operations are aborted after `(l-1)` bytes or when finding a newline.
  A valid pointer is returned on success, `NULL` on error.
- `int dkstream_puts(dk_stream_t *st, char *b);`
  writes a string from buffer `b` to the specified stream and returns `1` on success and `0` on error.
- `int`
  `dkstream_wb_word(dk_stream_t *st, dk_word w);`
  writes a 16-bit-word in network byte order to the stream.
- `int`
  `dkstream_wb_uword(dk_stream_t *st, dk_uword w);`
  writes an unsigned 16-bit-word in network byte order to the stream.

- `int dkstream_wb_dword(dk_stream_t *st, dk_dword w);`
  writes a 32-bit-double-word in network byte order to the stream.

- `int dkstream_wb_udword(dk_stream_t *st, dk_udword w);`
  writes an unsigned 32-bit-double-word in network byte order to the stream.

- `int dkstream_wb_string(dk_stream_t *st, char *str);`
  writes a string to a stream. The string length (including the finishing null-byte) is written as unsigned 32-bit-double-word in network byte order first followed by the string itself.

- `int dkstream_rb_word(dk_stream_t *st, dk_word *w);`
  reads a 16-bit-word from the stream, converts it to host byte order and saves it to the specified variable.

- `int dkstream_rb_uword(dk_stream_t *st, dk_uword *w);`
  reads an unsigned 16-bit-word from the stream, converts it to host byte order and saves it to the specified variable.

- `int dkstream_rb_dword(dk_stream_t *st, dk_dword *w);`
  reads a 32-bit-double-word from the stream, converts it to host byte order and saves it to the specified variable.

- `int dkstream_rb_udword(dk_stream_t *st, dk_udword *w);`
  reads an unsigned 32-bit-double-word from the stream, converts it to host byte order and saves it to the specified variable.

- `char * dkstream_rb_string(dk_stream_t *st);`
  reads a string from the specified stream and returns a pointer to the string.
  First the function reads the string length as unsigned 32-bit-double-word.
  Then it allocates memory for the string using `dk_new()`.
  The last step is to read the string into the new buffer.
  The buffer must be freed by using `dk_delete()` when it is no longer needed.

6.19.2 Writing handler functions

To use other I/O mechanisms you have to provide two functions:

- a handler function (sometimes referred to as callback function) and
- a function to open a resource and create a stream.

The `dkstream_...` functions use the `dk_stream_api_t` data type to pass arguments to the handler function and to obtain the result.
The function must be of type

```
void handler_function(dk_stream_api_t *apiptr);
```

The function has to check whether `apiptr` is a valid pointer. A pointer to the `dk_stream_t` originating the handler request can be retrieved as

```
dk_stream_t *s;
s = (dk_stream_t *)(apiptr->strm);
```

The handler again has to check whether this is a valid pointer or not. The assisting data can be retrieved from

```
s->data
```

The assisting data pointer is allowed to be `NULL` if the handler function can work without additional data.
The component `apiptr->cmd` contains an `int` value indicating the type of request. It can have the following values:

`DK_STREAM_CMD_TEST` The handler function is asked whether it can handle a certain request type.
This command is primarily used to check for a functionality like `fgets()` reading until an end of line occurs. See table 5.

Table 5: DK_STREAM_CMD_TEST

Component	Contents
`(apiptr->params).cmd`	The command to check for. An `int` value as used in the request type.
`apiptr->return_value`	`1` on success (command can be handled), `0` on error (can not handle command)

DK_STREAM_CMD_AT_END The handler function must perform a check whether the end of readable data is reached (no more data available). See table 6.

Table 6: DK_STREAM_CMD_AT_END

Component	Contents
`apiptr->return_value`	`1` when end of data reached, `0` when there is still data available.

DK_STREAM_CMD_FGETS We have to read a line of text into a buffer. See table 7.

Table 7: DK_STREAM_CMD_FGETS

Component	Contents
`(apiptr->params).buffer`	Start adress of buffer to fill.
`(apiptr->params).length`	Buffer length in bytes.
`apiptr->return_value`	`1` on success, `0` on error.

DK_STREAM_CMD_RDBUF The handler function has to read data from the stream into a buffer.

Table 8: DK_STREAM_CMD_RDBUF

Component	Contents
`(apiptr->params).buffer`	Start adress of buffer to fill.
`(apiptr->params).length`	Buffer length in bytes.
`apiptr->return_value`	`1` on success (any data was read), `0` on error.
`(apiptr->results).used`	Number of bytes successfully read, valid only when return_value indicated success.

DK_STREAM_CMD_FPUTS The handler function has to write a string.

Table 9: DK_STREAM_CMD_FPUTS

Component	Contents
`(apiptr->params).buffer`	Start adress of string.
	. . . to be continued

Continuation	
`apiptr->return_value`	`1` on success, `0` on error.
`(apiptr->results).used`	Number of bytes written, valid only if return_value indicates success.

`DK_STREAM_CMD_WRBUF` The handler has to write data from a buffer to the stream. See table 10.

Table 10: DK_STREAM_CMD_WRBUF

Component	Contents
`(apiptr->params).buffer`	Start adress of buffer.
`(apiptr->params).length`	Buffer length in bytes.
`apiptr->return_value`	`1` on (partial) success, `0` on error.
`(apiptr->results).used`	Number of bytes successfully written. Only valid if return_value indicates success.

`DK_STREAM_CMD_FLUSH` The handler function should perform an intermediate flush if possible. I/O-operations on files and gezipped files can perform flushes before the end of data is reached.
Some encoding mechanisms (i.e. ASCII85 as used in PS level 3) handle fixed block sizes only (except the last block). In these cases intermediate flushes must not be executed.

Table 11: DK_STREAM_CMD_FLUSH

Component	Contents
`apiptr->return_value`	`1` on success, `0` on error.

`DK_STREAM_CMD_FINAL` The handler function must perform a final flush. No further data is to be written.
Encoding mechanisms dealing with fixed block sizes can now handle the possibly incomplete last block. See table 12.

Table 12: DK_STREAM_CMD_FINAL

Component	Contents
	. . . to be continued

	Continuation
`apiptr->return_value`	`1` on success, `0` on error.

`DK_STREAM_CMD_REWIND` The stream must be rewound to set the current stream position to the start of stream.

Table 13: DK_STREAM_CMD_REWIND

Component	Contents
`apiptr->return_value`	`1` on success, `0` on error.

`DK_STREAM_CMD_FINISH` The handler function must release all resources which were allocated by the `dkstream_open_...` function. This is invoked as a part of `dkstream_close()`.

Table 14: DK_STREAM_CMD_FINISH

Component	Contents
`(apiptr->params).buffer`	Start adress of buffer to fill.
`(apiptr->params).length`	Buffer length in bytes.
`apiptr->return_value`	`1` on success, `0` on error.

See `file_stream_function()` in `dkstream.c` as an example.
The `dkstream_open_...` function must open/allocate I/O ressources. This allocation process must result in one variable.
Now `dkstream_new` can be called, a pointer to the variable and a pointer to the appropriate handler function must be provided.
During `dkstream_close()` a call to the handler function is made with command `DK_STREAM_CMD_FINISH`.
This is the point to close/deallocate the ressources before `dkstream_close()` calls `dkstream_delete()`.
See `dkstream_openfile()` in module `dkstream.c` as an example.

Now the new I/O system can be used like in the program skeleton:

```
dk_stream_t *st;
/*
  The next step allocates ressources for
  the I/O source/destination and sets up
  a new dk_stream_t variable.
*/
st = dkstream_open_...(...);
/*
  We must check for success.
*/
if(st) {
  /*
    Here we can do I/O operations.
  */
  /*
    Finally we must release I/O ressources
    and deallocate the memory for the dk_stream_t.
  */
  dkstream_close(st);
  st = NULL;
}
```

6.20 dkof - output filtering

6.20.1 Overview

Some file formats – i.e. PostScript and PDF – can use compressed and encoded data, different compression and encoding methods can be combined.
This module can be used to create compressed and encoded data.
The interface to the *writing* application is an expanded dk_stream_t API. The dk_stream_t represents the filter pipeline consisting of different filter cells. Each cell implements one compression or encoding method. If data is written to the dk_stream_t it goes to the top level cell first. This applies the first filtering method, i.e. flate compression. Output from this cell goes as input to the cell below, i.e. an ASCII85 encoding cell. Output from this cell goes downward, i.e. into a buffering cell collecting data to write buffers of a given size. Output from cell 0 (the bottom cell) goes to a target dk_stream_t.

6.20.2 Functions

The following functions are available:

- dk_stream_t *dkof_open(dk_stream_t *target, size_t nof);
  creates a new filter stream and returns a pointer to it. The *target* argument is the target stream responsible for writing/saving the encoded data. The *nof* argument is the number of filter cells needed.

- void dkof_close(dk_stream_t *str);
  closes a filter stream created by dkof_open(). The target filter stream is *not* closed.

- int dkof_set(dk_stream_t *str, size_t n, int what);
  sets a filter cell method. *str* is the filter stream returned by dkof_open(). *n* is the number of the filter cell to set up, it must be in the range $0 \ldots nof - 1$. Cell $nof - 1$ is the filter first applied to the data stream, cell 0 is applied last. *what* selects the filter cell type, it can be one of the following:

  - DK_OF_TYPE_FLATE
    flate compression.
  - DK_OF_TYPE_ASCII85
    ASCII-85-encoding.
  - DK_OF_TYPE_BUFFERED
    a buffering filter, writing data in buffers of 512 bytes.

- int dkof_start_chunk(dk_stream_t *str);
  starts a new data chunk. The filter cells are initialized.
- int dkof_end_chunk(dk_stream_t *str);
  finishes a data chunk. All filter cells are flushed, associated data is closed.
- void dkof_set_crnl(dk_stream_t *str, int v);
  enables/disables – depending on *v* – the CR/NL sequence instead of simple newline. This is used by the ASCII85 encoding output.

6.20.3 Usage

```
  dk_stream_t *os1, *os2;
  os1 = dkstream_for_file(stdout);      /* open real output stream */
  if(os1) {
    os2 = dkof_open(os1, 3);            /* create filter stream */
    if(os2) {
      dkof_set_crnl(os2,1);
      if(dkof_set(os2,0,DK_OF_TYPE_BUFFERED)) {
        if(dkof_set(os2,1,DK_OF_TYPE_ASCII85)) {
  if(dkof_set(os2,2,DK_OF_TYPE_FLATE)) { {
    dkstream_puts(os1, "%!PS-Adobe-3.0 EPSF-3.0\r\n");
    /* ... */
    dkstream_puts(os1, "colorimage\r\n");
    if(dkof_start_chunk(os2)) {
      unsigned char buffer[3];
      for(y = 0; y < height; y++) {
        for(x = 0; x < width; x++) {
  buffer[0] = get_red(x,y);
  buffer[1] = get_green(x,y);
  buffer[2] = get_blue(x,y);
  if(dkstream_write(os2, buffer, 3) != 3) {
    /* issue error message */
  }
}
      }
      if(dkof_end_chunk(os2)) {
        /* ... */
dkstream_puts(os1, "showpage\r\n");
      }
    }
  }
```

```
}
      }
      dkstream_close(os2);
    }
    dkstream_close(os1);
  }
```

6.21 dkcp - Dealing with codepages

6.21.1 Overview

Some systems need to deal with codepages. On Windows writing the german umlaut ö to a file requires to put another character than writing the same umlaut into a console window.
Codepages can be used for translation, so one can use the same message string to write to multiple destinations.
Codepages can be read from file or from somewhere else (i.e. network) using the dk_stream_t interface.
Once a codepage has been created and set up it can be used to convert characters for a specific output.

6.21.2 Functions

The following functions are available in the module:

- `unsigned char *dkcp_open DK_P1(dk_stream_t *,st)`
  allocates memory for an array of unsigned characters and initializes it by reading the codepage information from the stream.
  The array must be freed after usage by calling either dk_delete() or dkmem_free() on it.

- `unsigned char dkcp_convert DK_P2(unsigned char *, codepage, unsigned char, c)`
  converts one character using the specified codepage.

- `void dkcp_fputs DK_P3(FILE *, out, unsigned char *, cp, char *, str)`
  writes a string *str* to a file *out* using the codepage *cp*.

6.21.3 Codepage file structure

The codepage file is read line by line. A raute character is the beginning of a comment which is finished by the end of line. Each line contains one conversion rule. Conversion rules consist of two codes: the original character code and the destination character code. The original character code is the character we want to print in decimal notation. The destination character code is the decimal notation for the byte we must write to show the character. If the destination can not show the character, the destination character code is a minus sign.

6.21.4 Example

The example file cp.850 shows how to change character encoding when printing strings to a MS-DOS console.

```
#
# Character encoding changes for output on
# MS-DOS-prompt
#
161     173
162     -
163     156
164     207
165     190
166     221
167     245
168     249
169     184
170     -
171     174
172     175
173     -
174     169
175     238
176     -
177     241
178     253
179     252
```

6.22 dksto - Sorted and unsorted data storage

6.22.1 Overview

The `dksto` module contains data types and functions to store data in sorted and unsorted containers.
Unsorted containers are implemented as double-linked lists, sorted containers can use both double-linked lists and AVL-trees.
Iterators can be used to traverse the containers.
The following functions are available:

- `dk_storage_t *dksto_open(int pathlen);`
  allocates memory for a new container and returns a pointer to the new variable.
  For sorted containers using AVL-tree implementation `pathlen` specifies the maximum path length in the tree.
  The following constants can be used to create appropriate maximum path lengths and numbers of elementes:

DK_STO_SIZE_HUGE (0)	1536	exceeds 1.30699e+308
DK_STO_SIZE_LARGE (1)	1024	1.17987e+214
DK_STO_SIZE_MEDIUM (2)	512	1.17534e+107
DK_STO_SIZE_SMALL (3)	128	6.59035e+26
DK_STO_SIZE_TINY (4)	64	2.77779e+13

  For unsorted data storage use `DK_STO_SIZE_TINY`.
  The container must be freed by calling `dksto_close()` when it is not longer needed.

- `dk_storage_t *dksto_open(dk_storage_t *st);`
  closes a container and releases the resources allocated for the container including any iterators.
  The data objects referenced by the container are not freed.

- `int dksto_set_eval_c(`

```
dk_storage_t *st, dk_fct_eval_c_t *f, int cr
);
int dksto_set_eval_uc(
dk_storage_t *st, dk_fct_eval_uc_t *f, int cr
);
int dksto_set_eval_s(
dk_storage_t *st, dk_fct_eval_s_t *f, int cr
);
int dksto_set_eval_us(
dk_storage_t *st, dk_fct_eval_us_t *f, int cr
);
int dksto_set_eval_i(
dk_storage_t *st, dk_fct_eval_i_t *f, int cr
);
int dksto_set_eval_ui(
dk_storage_t *st, dk_fct_eval_ui_t *f, int cr
);
int dksto_set_eval_l(
dk_storage_t *st, dk_fct_eval_l_t *f, int cr
);
int dksto_set_eval_ul(
dk_storage_t *st, dk_fct_eval_ul_t *f, int cr
);
int dksto_set_eval_f(
dk_storage_t *st, dk_fct_eval_f_t *f, int cr
);
int dksto_set_eval_d(
dk_storage_t *st, dk_fct_eval_d_t *f, int cr
);
int dksto_set_comp(
dk_storage_t *st, dk_fct_comp_t *f, int cr
);
```

set up evaluation or comparison functions for sorted storage.
These functions must be called before the first object is added into the container. Evaluation and comparison functions are explained later.

- `int dksto_use_trees(dk_storage_t *st, int ok);`
  allows the usage of AVL-trees for the container (`ok != 0`) or denies it.
  This function must be called before the first object is added into the container.
  By default it is allowed to use trees. The AVLTREE environment vaiable

can be set to `yes` or `no` to change the default setting.

- `int dksto_add(dk_storage_t *st, void *obj);`
  stores an object reference in the container.

- `int dksto_remove(dk_storage_t *st, void *obj);`
  removes an object reference from the container.

- `dk_storage_iterator_t * dksto_it_open(dk_storage_t *st);`
  creates a new iterator on the container `st` and returns a pointer to it. The iterator should be freed by `dksto_it_close()` when it is not longer needed.
  Note: When `st` is closed by `dksto_close()` all iterators connected to it will be closed automatically.

- `void dksto_it_close(dk_storage_iterator_t *it);`
  closes an iterator and deallocates it's ressources.

- `void *dksto_it_find_exact( dk_storage_iterator_t *it, void *o);`
  searches in the container for the given object reference and returns a pointer or `NULL`.
  In sorted containers subsequent calls to `dksto_it_next()` will return pointers to objects "equal to" or "larger" than the given object.

- `void * dksto_it_find_like( dk_storage_iterator_t *it, void *o );`
  searches for an object with the same evaluation as the given object.
  In sorted containers subsequent calls to `dksto_it_next()` will return pointers to objects "equal to" or "larger" than the given object.

- `void *dk_sto_it_next(dk_storage_iterator_t *it);`
  can be used to traverse a container. It returns a pointer to the "next" object.

- `void *dksto_it_reset(dk_storage_iterator_t *it);`
  resets the iterator so `void *dk_sto_it_next();` finds the first element.

6.22.2 An example

Imagine you want to create data sets for employees, containing name, first name and age for each one.
You might use data structures as:

```
typedef struct {
  char *name; char *fname; unsigned age;
} Person;
```

We assume to have two functions available to create and destroy data sets:

```
Person
*new_person(char *name, char *fname, unsigned age);

void
delete_person(Person *person);
```

To sort data by name, first name and age you will need the following comparison and evaluation functionality:

- compare two data sets by name,
- compare a data set against a string containing a name,
- compare two data sets by first name,
- compare a data set against a string containing a first name and
- evaluate a data set by retrieving the age.

The first four items in the list are comparisons, they can be handled in one C-function by passing different comparison criteria.
Item 5 requires an evaluation function:

```
int compare_fct(void *p1, void *p2, int criteria)
{
 int back = 0;
 /*
  * make sure we have valid pointers
  */
 if(p1 && p2) {
 switch(criteria) {
  case 0: {
   /* compare 2 data sets by name */
```

```
   back =
   strcmp(((Person *)p1)->name, ((Person *)p2)->name);
  } break;
  case 1: {
   /* compare a data set against a given name */
   back =
   strcmp(((Person *)p1)->name, (char *)p2);
  } break;
  case 2: {
   /* compare 2 data sets by first name */
   back =
   strcmp(((Person *)p1)->fname, ((Person *)p2)->fname);
  } break;
  case 3: {
   /* compare a data set against a given first name */
   back =
   strcmp(((Person *)p1)->fname, (char *)p2);
  } break;
 }
 }
 return back;
}

unsigned get_age(void *p, int criteria)
{
 unsigned back = 0;
 if(p) {
  back = ((Person *)p)->age;
 }
 return back;
}
```

Now we can establish four containers (three for sorted storing, one for unsorted storing.

```
dk_storage_t *s1, *s2, *s3, *s4;
dk_storage_iterator_t *i1, *i2, *i3, *i4;
s1 = dksto_open(0);
s2 = dksto_open(0);
s3 = dksto_open(0);
s4 = dksto_open(DK_STO_SIZE_TINY);
if(s1 && s2 && s3 && s4) {
```

```
 i1 = dksto_it_open(s1);
 i2 = dksto_it_open(s2);
 i3 = dksto_it_open(s3);
 i4 = dksto_it_open(s4);
 /* sorted by name */
 dksto_set_comp(s1,compare_fct,0);
 /* sorted by first name */
 dksto_set_comp(s2,compare_fct,2);
 /* sorted by age */
 dksto_set_eval_u(s3,get_age,0);
 /* ... use the containers ... */
}
if(s1) dksto_close(s1); if(s2) dksto_close(s2);
if(s3) dksto_close(s3); if(s4) dksto_close(s4);
```

The code to insert data into the containers is as follows:

```
Person *p;
while(...) {
 p = new_person(...);
 if(p) {
  if(dksto_add(s4,p)) {
   dksto_add(s1,p);     /* perform checks here too */
   dksto_add(s2,p);
   dksto_add(s3,p);
  } else {
   delete_person(p);
   fatal("NOT ENOUGH MEMORY TO STORE PERSON!");
  }
 }
}
```

To find user JOE AVERAGE and all first names coming after JOE use

```
char *name = "Joe";
Person *p;
/*
  Note criteria 3 for comparison
  of data set against pure name
*/
p = (Person *)dksto_it_find_like(i2,name,3);
if(p) {
```

```
 printf(
  "Name %s, First name: %s, Age %u\n",
  p->name, p->fname, p->age
 );
 while((p = (Person *)dksto_it_next(i2)) != NULL) {
  printf(
   "Name %s, First name: %s, Age %u\n",
   p->name, p->fname, p->age
  );
 }
}
```

6.23 dkstt - String tables

String tables are key/value-pairs in binary form. They can be used to localize an application. The `dk_stt_t` data type is used to represent a string table.
The following functions are available in the module:

- `dkstt_open(dk_stream_t *st);`
  reads a string table from the specified stream, allocates all memory necessary and returns a pointer to the string table (or `NULL`).
  The stream must be open and remains open.

- ```
  char *dkstt_find(
  dk_stt_t *s, char *key, char *def
  );
  ```
  searches for an entry matching the given key in the string table and returns a pointer to the value (or `NULL`).

- `dkstt_close(dk_stt_t *s);`
  closes a string table and deallocates all memory used by it. All pointers returned by `dkstt_find()`-calls to this string table become invalid and can no longer be used.
  ```

6.24 dkapp - Application

The `dkapp` module encapsulates things needed by full-featured command line applications like:

- Preferences handling.
  Preferences are persistent key/value pairs of strings residing in configuration files or registry entries.
  Preferences can be overwritten by command line arguments. The application can exclude scopes (command line preferences, preferences set by the application itself, user set preferences, administrator set preferences) when retrieving preferences.
  Preferences are valid in scopes, a scope consists of

  – a username,
  – an application name and
  – a host name.

  A scope is valid if the scopes username matches the current username, the application name matches the current application name and the scopes host name matches the current host name. If we have - for instance - the user name `joe`, the application name `dosomething` and the host name `pc` the following scopes are valid:

  – */*/*
  – */*/pc
  – */dosomething/*
  – */dosomething/pc
  – joe/*/*
  – joe/*/pc
  – joe/dosomething/*
  – joe/dosomething/pc

  The last scope has highest priority, preferences defined in this scope overwrite all others.

- File search depending on the users preferred language and region.
  This is used to read the string tables and help text files matching the users preferences.

- Logging.
  Log output can be written to
  - standard output (stdout),
  - standard error output (stderr),
  - a logfile or
  - the `syslog` system (if available on the given host).

  A log level is assigned to each message logged indicating the "heaviness" of the message.
  Preferences can be used to control which "heaviness" is required for each of the log destinations.

- Getting names for temporary files.
  Normally a call to `tmpnam(NULL)` returns a pointer to a file name residing in a directory like `/var/tmp` specifically created to store temporary files. On W* systems this returns a filename in the root directory of the current drive. RANF because write access to everything outside the temporary and home directories should be denied for normal (stupid) users.

The following functions are available in the module:

- `dk_app_t *`
  `dkapp_open DK_PR((int argc, char *argv[]));`
  allocates memory for a new `dk_app_t` and returns a pointer to it.
  The `dk_app_t` variable is initialized using the `main()` functions `argc` and `argv` arguments:
  - A copy of the `argv` arguments is made.
  - All arguments starting with a leading `--/` are interpreted as preferences and removed from the copy.
  - The application tries to retrieve user name, host name and application name.
    The application name is derived from `argv[0]` by extracting the file name part and removing the file type suffix (if available).
  - The application retrieves information about the users home directory and the directory for temporary files.
  - The filename for the file executed is searched.
  - The directory containing the file executed is searched. This directory is assumed to be the application directory and the directory for shared files.

- If the directory ends on `bin` the directory for shared files is estimated by replacing the `bin` by `lib`. The application name is appended to build a new name for the application directory.
- The `dk_storage_t` components for preferences management are initialized.
- The preferences file `appdefault` is read from the shared directory on systems without a registry.
- The preferences file `appdefault.`*`application`* is read from the shared directory on systems without a registry.
- The preferences file
  * `/etc/appdefaults` or
  * `C:/ETC/ALL.DEF`

  is read on systems without a registry.
- The preferences file
  * `/etc/appdefaults.`*`application`* or
  * `C:/ETC/`*`application`*`.DEF`

  is read on systems without a registry.
- The preferences file `all` is read from `$HOME/.defaults` on systems without a registry.
- The preferences file `$HOME/.defaults/`*`application`* is read on systems without a registry.
  On systems where leading dots in filename are not allowed the files are searched in `$HOME/defaults`.
  If preferences are defined multiple time the setting with the highest priorized scope is valid.
  If the highest scope occurs several times the latest setting is valid.
- The preference `/dir/app` is looked up. If this preference is defined it's value is used for the application directory.
- The preference `/dir/shared` is looked up. If this preference is defined it's value is used for the shared directory.
- The preference `/dir/tmp` is looked up. If this preference is defined it's value is used for the temporary directory.
- A subdirectory is created in the temporary directory to keep this applications temporary files.
  The subdirectories name is consists of the PID as hexadecimal string

and the suffix ".TMP".
On systems without the `getpid()` functions another unique number is choosed.
Note: There is a possible race condition because there is some time between the file name check and the directory creation. To prevent other users from exploiting this it is recommended to set `/dir/tmp` to point to a directory inside the users home directory.

– Logging to file is initialized.
The following preferences can be used to set up logging to file:
  * `/log/file/name`
  The name of the logfile.
  * `/log/file/level`
  The minimum "heaviness" for log message, can be "none" (no output), "panic", "fatal", "error", "warning", "info", "progress" or "debug".
  Only messages having at least the required heaviness are logged to file.
  The default is "warning".
  * `/log/file/keep`
  The keep level.
  To prevent programs from filling the hard disk log files are deleted at the programs end unless there was at least one message of a given heaviness.
  The default is "error".
  * `/log/file/time`
  This boolean flag specifies whether the current time is written for each log message or not.
  The default is "yes".
  * `/log/file/split`
  This boolean flag is of interest only when time stamps are saved together with log messages. It specifies whether the time stamps are written to separate lines or not.
  The default is "yes".

– Logging to `stdout` is set up using the preferences
  * `/log/stdout/level`,
  * `/log/stdout/time` and
  * `/log/stdout/split`.

– Logging to `stderr` is set up using the preferences

    * `/log/stderr/level`,
    * `/log/stderr/time` and
    * `/log/stderr/split`.
  - If `syslog` is available, `openlog()` is called.
  - The PID of the current process is retrieved.
  - The users preferred language is retrieved from the command line preference `/ui/lang`, the environment variable `LANG` or the `/ui/lang` preference from any other source. The first value found is used.
  - Internal string tables are read.
  - Log messages about the program startup are issued.

For boolean flags the following values (case-insensitive) indicate "true":

- `1`,
- `yes`,
- `y`,
- `on`,
- `ok and`
- `true`.

In the preferences defining the application directory, the shared directory and the log file name the following macros (case-sensitive) are allowed:

- `$(app.name)`
  The application name.
- `$(temp.dir)`
  The directory for temporary files.
- `$(user.name)`
  The users login name.
- `$(user.home)`
  The users home directory.
- `$(user.uid)`
  The users UID.
- `$(user.gid)`
  The users GID.
- `$(user.euid)`
  The effective UID.

  - `$(user.egid)`
    The effective GID.
  - `$(host.name)`
    The host name (short form).
  - `$(host.domain)`
    The hosts DNS domain.
  - `$(process.pid)`
    The process ID of the running process.
  - `$(process.ppid)`
    The PID of the parent process.
  - `$(process.pgid)`
    The process group ID of the current process.

- ```
  dk_app_t *dkapp_open_ext1(
  int argc, char *argv[], char *g, char *etc,
  int sil, int nostdout
  );
  ```
  does the same as `dkapp_open()`, it is extended version 1.
  The `g` parameter is the application group name (package name), this parameter may be `NULL` if no package name is used.
  The `etc` parameter points to the system configuration directory (by default "/etc" on U*x systems), one should use a variable defined via the `configure` mechanism here. This parameter may be `NULL` if not needed.
  The `sil` parameter can be used to make the application behave silently.
  The `nostdout` parameter can be used to deny writing of log messages to `stdout` and `stderr`. This is for GUI applications which do not have a console attached by default.
  In general one should prefer to use `dkapp_open_ext1()` instead of

  - `dkapp_open()` and
  - `dkapp_set_groupname()` or `dkapp_set_silent()`

  because `dkapp_open_ext1()` can use the package name and the silence flag during application startup.

- `int dkapp_set_groupname(dk_app_t *a, char *name);`
  sets the application group name (package name) for an application.
  Note: It is recommended to use `dkapp_open_ext1()` to set the group name.
  ```

- `int dkapp_set_silent(dk_app_t *a, int flag);`
  sets the application to silent behaviour (the minimum log levels are increased so no log messages are written) if $flag \neq 0$.
  If `a` is `NULL` a module specific variable is set to make all applications created by subsequent `dkapp_open()` calls work silently.
  Note: It is recommended to use `dkapp_open_ext1()` to set the application in silent mode from the beginning.

- ```
  void dkapp_set_source_filename(
  dk_app_t *a, char *n
  );
  ```
  sets the filename to be issued with the next log messages.
  This function and the following one can be used to create messages containing filename and line number in the source file causing the error.

- ```
  void dkapp_set_source_lineno(
  dk_app_t *a, unsigned long l
  );
  ```
  sets the line number to be issued with the next log message.

- ```
  int dkapp_log_msg(
  dk_app_t *a, int p, char **msg, int az
  );
  ```
  writes a log message of priority `p`. The priority can be

  - `DK_LOG_LEVEL_PANIC`
    in panic conditions,
  - `DK_LOG_LEVEL_FATAL`
    for fatal errors,
  - `DK_LOG_LEVEL_ERROR`
    for errors,
  - `DK_LOG_LEVEL_WARNING`
    for warnings,
  - `DK_LOG_LEVEL_INFO`
    for normal messages,
  - `DK_LOG_LEVEL_PROGRESS`
    to show that the program is still working or
  - `DK_LOG_LEVEL_DEBUG`
    for debug output.
  ```

The message consists of several strings, `msg` points to an array of string pointers, `az` is the number of array elements.

- ```
  int dkapp_get_pref(
  dk_app_t *a, char *key,
  char *buf, size_t lgt, int ex
  );
  ```
  retrieves the preference value for the given `key` and stores the value in buffer `buf` of size `lgt`.
  The parameter `ex` specifies which preference sources are *excluded* from the lookup process, specify

  – `DK_APP_PREF_EXCL_CMD`
    for command line options,
  – `DK_APP_PREF_EXCL_PROG`
    for programs own settings,
  – `DK_APP_PREF_EXCL_USER`
    for the users preferences files and
  – `DK_APP_PREF_EXCL_SYSTEM`
    for preferences set by the administrator.

  To exclude mulitple data sources or-combine the values.
  A value `0` uses all data sources for lookup.
  The function returns `0` if no value was found, otherwise another value.

- ```
  int dkapp_set_pref(
  dk_app_t *a, char *key, char *value
  );
  ```
  sets a preference value.

- ```
  int dkapp_transform_string(
  dk_app_t *app,char *dest,size_t sz,char *src
  );
  ```
  transforms the string `src` to buffer `dest` of size `sz` and replaces `$(...)`-macros by their values.
  In addition to the macros mentioned above the following macros can be used:

  – `$(app.dir)`
    The application directory.
  – `$(shared.dir)`
    The shared directory.
  ```

- `int dkapp_find_cfg(`
  `dk_app_t *a, char *name, char *buffer, size_t sz`
  `);`
  searches for a file having the given `name` and writes the full filename into the `buffer` of length `sz`.
  The function can be used to search for compressed and localized versions of configuration and resource files.
  It searches in the Windows directory and subdirectories (on W* systems) and in the system configuration directory and subdirectories.
  The directories are checked for a subdirectory structure matching the users preferred language, region and encoding.
  See section 10.4 on page 141 for details.

- `int dkapp_find_file(`
  `dk_app_t *a, char *name, char *buffer, size_t sz`
  `);`
  searches for a file having the given `name` and writes the full filename into the buffer `buffer` of length `sz`.
  This function can be used to search for compresseed and localized versions of resource and configuration files.
  It searches in the current directory, in the application directory, the application group directory, the shared directory, the Windows directory and subdirectories (on W* systems) and in the system configuration directory and subdirectories[1].
  Each of these directories is checked for a subdirectory structure taking care of languages, regions and encodings.
  See section 10.3 on page 138 for details.

- `char *dkapp_find_string(`
  `dk_app_t *a, char *t, char *k, char *d`
  `);`
  searches for a string table entry with key `k` in table `t` and returns a pointer to the string found or the default string pointer `d`.

- `void dkapp_find_multi(`
  `dk_app_t *app, dk_string_finder_t *f, char *table`
  `);`
  searches for multiple strings at once and sets up pointers.

- `void dkapp_help(`

[1] named as the application or the application group

```
dk_app_t *a, char *filename, char **def_strings
);
```
searches for a localized help file `filename` and prints this to stdout. If the file is not found, the `NULL`-terminated string array `def_strings` is printed.

- ```
  dk_stream_t *dkapp_read_file(
  dk_app_t *app, char *filename
  );
  ```
  searches for the given `filename` (localized, possibly compressed) and opens it for read access.

- ```
  dk_stream_t *dkapp_write_file(
  dk_app_t *app, char *filename
  );
  ```
  tries to open the file `filename` for write access and returns a `dk_stream_t` pointer on success.

- ```
  int dkapp_get_argc(
  dk_app_t *a
  );
  char **dkapp_get_argv(
  dk_app_t *a
  );
  ```
  can be used to retrieve the command line arguments without command line preferences settings.

- ```
  int dkapp_tmpnam(
  dk_app_t *a, char *buffer, size_t sz
  );
  ```
  fills write a pathname for a temporary file into the buffer.

- ```
  void dkapp_unconfigure(
  dk_app_t *app
  );
  ```
  causes the application to delete the applications preference file or the registry key when `dkapp_close();` is called.

- ```
  void dkapp_err_traverse_dir(
  dk_app_t *app, char *name
  );
  ```
  issues an error message that the given directory cannot be traversed.

- `void dkapp_err_stat_failed(`
  `dk_app_t *app, char *name`
  `);`
  issues an error message that no information is available about the given file.

- `void dkapp_err_cwd(`
  `dk_app_t *app`
  `);`
  issues an error message telling that the current working directory cannot be estimated.

- `void dkapp_err_memory(`
  `dk_app_t *app, size_t elsize, size_t nelem`
  `);`
  issues an error message that the application failed to allocate memory for `nelem` elements of size `elsize`.

- `dkapp_err_matchfile(`
  `dk_app_t *app, char *name`
  `);`
  issues an error message that there is no file matching the name pattern.

- `void dkapp_err_matchdir(`
  `dk_app_t *app, char *name`
  `);`
  issues an error message that there is no directory name matching the pattern.

- `void dkapp_err_fopenr(`
  `dk_app_t *app, char *name`
  `);`
  issues an error message that the program failed to open the file for read access.

- `void dkapp_err_fopenw(`
  `dk_app_t *app, char *name`
  `);`
  issues an error message that the program failed to open the file for write access.

- `void dkapp_err_fwrite(`
  `dk_app_t *app, char *name`
  `);`
  issues an error message that there was an error during a write operation.

- ```
  void dkapp_err_fread(
  dk_app_t *app, char *name
  );
  ```
  issues an error message that there was an error during a read operation.

- ```
  void dkapp_close(
  dk_app_t *a
  );
  ```
  closes the application and releases all memory allocated by the `dk_app_t` variable.
  This should be a programs last action before exiting.

- ```
  dk_stream_t *dkapp_stream_openfile(
  dk_app_t *a, char *n, char *m
  );
  dk_stream_t *dkapp_stream_opengz(
  dk_app_t *a, char *n, char *m
  );
  dk_stream_t *dkapp_stream_openbz2(
  dk_app_t *a, char *n, char *m
  );
  FILE *dkapp_fopen(
  dk_app_t *a, char *n, char *m
  );
  ```
  are used to open files. The command line preferences of the application `a` control which security checks are performed if the file with name `n` is opened for writing (the checks are performed additionally to the platform-dependant permission checks). By default writing is denied if `n` is a symbolic link and at least one of the following conditions is fullfilled:

  – The owner of the link is not the owner of the link target file.
  – The link is placed in a group writable directory.
  – The link is placed in a world writable directory.

  One can skip – for example the first check – by typing

  ```
  <app> --/sec/ign/link-owner=true
  ```

  See table 16 on page 136 for a list of preferences to skip security checks.
  If a check fails the file is not opened and a log message is issued.
  ```

6.25 dktcpip - TCP/IP networking

This module is for the client side only. On the server side you should directly deal with BSD sockets, TLI/XTI or Windows sockets to get the best performance. The `dktcpip` module contains support for the following data types:

- `dk_ip_addr_t`
  stores an address.
- `dk_tcpip_t`
  contains data for a transport endpoint.

Before using any of the networking functions the application must call
`int dktcpip_start(void);`
to initialize TCP/IP networking. If you are done with networking call
`int dktcpip_end DK_PR((void));`
to clean up the networking system.
Between the two function calls you can

- create transport endpoints,
- configure transport endpoints,
- bring transport endpoints up (connect them to a peer),
- transfer data via transport endpoints,
- bring transport endpoints down and
- destroy transport endpoints.

The following functions are in the module:

- `int dktcpip_start(void);`
  connects the process to the TCP/IP-networking subsystem and returns `1` on success and `0` on error.
  On *nix systems the function simply returns `1`, on W* systems it calls `WSAStartup()`.
  If the function returns successfully the process must call
  `int dktcpip_end(void);`
  before terminating.
- `int dktcpip_end(void);`
  disconnects the process from the TCP/IP-networking subsystem. On W* systems it calls `WSACleanup()`, on *nix systems it simply returns `1`.

- `dk_tcpip_t *dktcpip_new(void);`
  creates a new transport endpoint and returns a pointer to the endpoint data or `NULL` on error.
  On success the endpoint must be destroyed by `dktcpip_delete()` when it is not longer needed.

- `void dktcpip_delete(dk_tcpip_t *p);`
  destroys a transport endpoint and releases the memory associated with it.

- ```
  dk_ip_addr_t *dktcpip_get_addr(
  dk_tcpip_t *p, int w
  );
  ```
  retrieves a pointer to one of four address buffers contained in the endpoint data.
  The parameter `w` determines which address is selected:

  - `DK_IP_ADDR_REMOTE_WISHED`
    the remote address we want to connect to.
    For connectionless transport this is the address to which we want to send.
  - `DK_IP_ADDR_REMOTE_FOUND`
    the remote address we are connected to.
    For connectionless transport this is the address from which we received the last datagram.
  - `DK_IP_ADDR_LOCAL_WISHED`
    the local address we want to bind to.
  - `DK_IP_ADDR_LOCAL_FOUND`
    the local address we are bound to.

  A valid pointer is returned on success, `NULL` on error.
  The `dktcpipaddr_set...` functions described below use this pointer.

- `int dktcpipaddr_set_ip_byname(`
  ```

```
dk_ip_addr_t *a, char *hn, dk_tcpip_t *p
);
int dktcpipaddr_set_ip_loopback(
dk_ip_addr_t *a
);
dktcpipaddr_set_ip_local(
dk_ip_addr_t *a, dk_tcpip_t *p
);
int dktcpipaddr_set_ip_any(
dk_ip_addr_t *a
);
```
are used to set the IP address part of the structure to

  - the IP address of a given host name `hn` (host name or IP address in dotted notation),
  - the local loopback interface,
  - the local IP address or
  - any IP address.

- ```
  int dktcpip_addr_set_port(
  dk_ip_addr_t *a, unsigned short min, unsigned short
  max
  );
  ```
  sets the port range. The `min` and `max` port number must be specified in *host notation.*
  If you want exactly one port number, use equal `min` and `max`.

- ```
  int dktcpip_set_connectionless(
  dk_tcpip_t *p, int flag
  );
  ```
  configures the endpoint for connectionless transport (UDP) if $flag \neq 0$ or connection-oriented transport (TCP) if $flag = 0$.
  By default the endpoints are set up for TCP.

- ```
  int dktcpip_set_timeout(
  dk_tcpip_t *t, double to
  );
  ```
  configures a timeout for the socket operations in microseconds.

- ```
  int dktcpip_set_nonblock(
  dk_tcpip_t *t, int fl
  );
  ```

configures the endpoint for non-blocking transport if $fl \neq 0$.

- `int dktcpip_set_reuse(`
  `dk_tcpip_t *t, int fl`
  `);`
  sets the `SO_REUSEADDR` option for the socket if $fl \neq 0$. This is usefull for TCP sockets only.

- `int dktcpip_set_broadcast(`
  `dk_tcpip_t *t, int fl`
  `);`
  sets the `SO_BROADCAST` option for the socket if $fl \neq 0$. This allows UDP sockets to send broadcast datagrams.

- `int dktcpip_set_keepalive(`
  `dk_tcpip_t *t, int fl`
  `);`
  configures the socket to set the `SO_KEEPALIVE` option to send periodic test messages to keep the socket alive if no data is transmitted.

- `int dktcpip_up(dk_tcpip_t *t);`
  tries to bring the endpoint up (make it ready to send and receive data).
  Before you do this the socket must be configured for connection-oriented or connectionless transport.
  The local wished address must be configured.
  For connection-oriented transport the remote wished address must be configured.
  If it is necessary to call
  - `dktcpip_set_timeout()`,
  - `dktcpip_set_reuse()`,
  - `dktcpip_set_broadcast()` or
  - `dktcpip_set_keepalive()`

  this should be done before calling `dktcpip_up()`.

- `int dktcpip_read(`
  `dk_tcpip_t *t, char *buf, size_t *lgt`
  `);`
  tries to read data from the network to a buffer. The `lgt` argument points to a variable containing the buffer size when invoking the function and the

number of used bytes after returning.
For connectionless transport the remote wished address must be configured before calling this function.

- `int dktcpip_respond(dk_tcpip_t *t);`
  sets the remote wished address to the remote found address.
  This is usefull for connectionless transport to send data to the same address we received the last datagram from.

- ```
  int dktcpip_write(
  dk_tcpip_t *t, char *buf, size_t *lgt
  );
  ```
  sends data. The `lgt` parameter points to a variable containing the buffer length when invoking the function and the number of bytes really sent when returning.
  For connectionless transport the remote wished address must be configured before invoking this function.

- `int dktcpip_closewrite(dk_tcpip_t *t);`
  indicates that the process makes no further write attempts on the endpoint.

- `int dktcpip_is_rdclosed(dk_tcpip_t *t);`
  checks whether the peer has finished writing, so we found end of input from network.

- `int dktcpip_down(dk_tcpip_t *t);`
  shuts down the endpoint.
  For connection-oriented transport `dktcpip_closewrite()` is called to have an orderly release, `dktcpip_read()` is called until `dktcpip_is_rdclosed()` returns true.

- ```
  int dktcpip_get_error_code(
  dk_tcpip_t *t, int cl
  );
  ```
  retrieves a code for the last error on the endpoint.
  If $cl \neq 0$ the error condition is cleared.

7 Administration

7.1 Installation

A `configure` script should be contained in every source package using the `dk` libraries.
A directory layout should be used like this:

- `${prefix}/bin`
  should be used for binary executables.
- `${prefix}/lib/myapp1`
  should be used to store additional files (i.e. string tables, help text files ...) supporting the `myapp1` executable program.

7.2 Preferences management

7.2.1 Preferences storage

If an application (i.e. `myapp1`) starts it first searches for the directory where the executable file came from. This directory is established as the application directory and as the shared directory.
If this directories last part is `bin` the `bin` in the shared directory name is replaced by `lib`.
If the shared directory contains a subdirectory matching the application name (`myapp1`) this subdirectory is established as the new application directory.
Before the preference files are read the command line arguments are inspected. If there are arguments starting with `--/...` they are treated as preference overwrites, removed from the argument list and inserted into the preferences storage as preference `/...`.
First the files

- `appdefaults` and
- `appdefaults.`*`application`* (here `appdefaults.myapp1`)

are read from the shared directory. These files should be used to set the `/dir/app` and `/dir/shared` preferences if your installation does not follow the directory layout explained above.
If the software is installed on a file server you can install network-wide preference settings here instead of creating files in each hosts `/etc` directory.
The next preferences files read are

- `/etc/appdefaults` (`C:/ETC/ALL.DEF on W* systems`) and

- `/etc/appdefaults.`*`application`* (`C:/ETC/`*`application`*`.DEF` on `W* systems`).

These files should be used to set preference defaults for all users.
Next the files `all` and *`application`* are read from the subdirectory `.defaults` (`defaults` on W* systems) in the users home directory. Users can establish preferences here.
The preferences files are only used on systems without a registry.

7.2.2 Configuration files example

We assume to have an application `myapp1` installed, so we have the binary file residing as `/network/apps/ours/bin/myapp1` and the additional files in directory `/network/apps/ours/lib/myapp1` and its subdirectories.
`/network/apps/ours/lib/appdefaults` might contain the following text:

```
#
# valid for all users, all applications on all hosts
# we could also write [*/*/*] instead
#
[*]
#
# set the shared directory and the
# application directory
#
/dir/shared = /network/apps/ours/lib
#
# see the macro in the next line
#
/dir/app    = /network/apps/ours/lib/$(app.name)
#
# use the local /tmp directory for temporary files
#
/dir/tmp    = /tmp
#
# set the language
#
/ui/lang    = de_DE
#
# Set up logging
#
```

```
/log/file/level = info
/log/file/keep = error
/log/file/name = $(app.name).$(process.pid).log
/log/file/time = off
#
# now we have special settings for a
# guest account named joe
# who speaks english
#
[joe]
/ui/lang    = en_US
#
# Application poorapp comes without support
# for german
#
[*/poorapp]
/ui/lang    = en_US
```

A user could write $HOME/.defaults/all like:

```
#
# This is valid for all applications on all
# hosts and could be written as [*/*]
#
[*]
#
# The language should be french
#
/ui/lang    = fr
#
# We keep temporary files under our control
#
/dir/tmp    = $(user.home)/tmp
#
# We want all the debug information if an
# error occured
#
/log/file/level = debug
/log/file/time  = on
/log/file/split = on
#
# The poorapp applications does not speak french
```

```
#
[poorapp]
/ui/lang    = en_US
```

The last two lines could be removed, in `$HOME/.defaults` we could create a file `poorapp` instead:

```
#
# on every host
#
[*]
#
# language english
#
/ui/lang = en_US
```

7.2.3 Preferences storage on 32-bit W* systems

Preferences are stored in 4 registry keys:

- `HKLM/Software/Dkapp`
- `HKLM/Software/Dkapp/`*`appname`*
- `HKCU/Software/Dkapp`
- `HKCU/Software/Dkapp/`*`appname`*

The keys in HKLM are valid for all users, the keys in HKCU are used for user-specific settings.
An administrator can use the HKLM-keys to provide default settings. Normal users need only read permissions to the HKLM-keys.
Values in `HKLM/Software/Dkapp` and `HKCU/Software/Dkapp` are general settings for all applications, values in `HKLM/Software/Dkapp/`*`appname`* and `HKCU/Software/Dkapp/`*`appname`* are for specific applications.
The registry value name consists of the preference scope and the preference name separated by colon. In the HKLM keys the scope consists of a user name (or wildcard `all`) and a host name (or wildcard `all`) separated by a slash. In the HKCU keys the scope simply consists of a host name.

Examples:
The value

```
"all/all:/ui/lang" = "de_DE"
```

in `HKLM/Software/Dkapp` sets the default language to german for all users on all hosts using all applications.

The value

```
"joe/all:/ui/lang" = "en"
```

in `HKLM/Software/Dkapp` sets the default language to english for user `joe` an all hosts using all applications.

The value

```
"all/all:/ui/lang" = "en"
```

in `HKLM/Software/Dkapp/app1` sets the language to english for all users on all hosts when using application `app1`.

If user `pedro` wants to have all applications in spanish he can set

```
"all:/ui/lang" = "sp"
```

in `HKCU/Software/Dkapp`. Because application `app1` is available in english only he needs an additional entry

```
"all:/ui/lang" = "sp"
```

in `HKCU/Software/Dkapp/app1`.

7.3 Recommended environment variables for W*32 systems

W*32 systems use different locations in the registry to store information about the current user, the home directory, the temporary directory etc.
Also for data storage different places on the hard disk are used depending on the W* version and the users language.
The following environment variables should be set on W*32 systems:

Variable	Contents
LOGNAME	The username of the user running the program.
HOME	The home directory of the user running the program.
TMPDIR	The directory to be used for storing temporary files.
COMPUTERNAME	The hostname.

8 Security

The following functions and modules should not be used in SUID/SGID programs, networking daemons or W*32 services:

- the `dkapp` module,
- `dksf_get_uname()`,
- `dksf_get_euname()`,
- `dksf_get_home()`,
- `dksf_get_ehome()`,
- `dksf_get_hostname()`,
- `dksf_get_tempdir()`,
- `dksf_get_domainname()`,
- `dksf_get_executable()`.

When using

- `dksf_getpid()`,
- `dksf_getppid()`,
- `dksf_getuid()`,
- `dksf_getgid()`,
- `dksf_geteuid()`,
- `dksf_getegid()`,
- `dksf_getpgrp()`,
- `dksf_getpgid()`,

it is recommended to use the appropriate `dksf_have_get...()` functions to check the system's support for `dksf_getpid()`...`dksf_getpgid()`.

9 Tutorial

9.1 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.

9.2 Introduction to the dkapp module

9.2.1 The simple Hello-world-program

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

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  printf("Hello world!\n");
  printf("Goodbye world!\n");
  exit(0);
  return 0;
}
```

9.2.2 Adding application support

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

```
#include <dkapp.h>

#include <stdio.h>
#if DK_HAVE_STDLIB_H
#include <stdlib.h>
#endif

dk_app_t *app = NULL;

static int success = 0;

int main(int argc, char *argv[])
{
  app = dkapp_open(argc, argv);
  if(app) {
    printf("Hello world!\n");
    printf("Goodbye world!\n");
    success = 1;
    dkapp_close(app); app = NULL;
  }
  success = (success ? 0 : 1);
  exit(success);
  return success;
}
```

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

```
./hello02
```

the expected output

```
Hello world!
Goodbye world!
```

appears.
The command

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

already shows some logging.
The `--/...` arguments can be used to override preference settings in the configuration 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.
```

9.2.3 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 `hell02`, `hell03`, `hello04` ....
The contents of the files should be as follows:

```
/dir/app          = /home/myname/src/libs/Docu/tutorial
/log/file/level   = debug
/log/file/keep    = error
/log/stderr/level = progress
```

Write the name of the `tutorial` directory you use instead of `/home/myname/src/libs/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`.

9.2.4 Internationalization

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

```
#include <dkapp.h>

#include <stdio.h>

#if DK_HAVE_STDLIB_H
#include <stdlib.h>
#endif

dk_app_t *app = NULL;

static int success = 0;

static char *messages[2];

static dk_string_finder_t finder[] = {
  { "hello",   &(messages[0]), "Hello world!" },
  { "goodbye", &(messages[1]), "Goodbye world!" },
  { NULL, NULL, NULL }
};

int main(int argc, char *argv[])
{
  app = dkapp_open(argc, argv);
  if(app) {
    dkapp_find_multi(app, finder, "h03msg");
    printf("%s\n", messages[0]);
    printf("%s\n", messages[1]);
    success = 1;
    dkapp_close(app); app = NULL;
  }
  success = (success ? 0 : 1);
  exit(success);
  return success;
}
```

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.

9.2.5 Logging

We change our program to `hello04.c` as follows:

```
#include <dklogc.h>
#include <dkapp.h>

#include <stdio.h>
#if DK_HAVE_STDLIB_H
#include <stdlib.h>
#endif

dk_app_t *app = NULL;

static int success = 0;

static char *messages[7];

static dk_string_finder_t finder[] = {
  { "/msg/pan",   &(messages[0]),
    "Test panic message!"
  },
  { "/msg/fat", &(messages[1]),
    "Test fatal error message!"
  },
  { "/msg/err", &(messages[2]),
    "Test error message."
  },
  { "/msg/wrn",  &(messages[3]),
    "Test warning message."
  },
  { "/msg/inf",  &(messages[4]),
    "Test informational message."
  },
  { "/msg/prg", &(messages[5]),
    "Test progress message."
  },
  { "/msg/dbg", &(messages[6]),
    "Test debug message."
  },
  { NULL, NULL, NULL }
};
```

```
static char *an_array_of_strings[] = {
  "Jim ",
  "and Joe",
  " are drinking beer.",
};

int main(int argc, char *argv[])
{
  app = dkapp_open(argc, argv);
  if(app) {
    dkapp_find_multi(app, finder, "h04msg");
    dkapp_log_msg(
      app, DK_LOG_LEVEL_PANIC,
      &(messages[0]), 1
    );
    dkapp_log_msg(
      app, DK_LOG_LEVEL_FATAL,
      &(messages[1]), 1
    );
    dkapp_log_msg(
      app, DK_LOG_LEVEL_ERROR,
      &(messages[2]), 1
    );
    dkapp_log_msg(
      app, DK_LOG_LEVEL_WARNING,
      &(messages[3]), 1
    );
    dkapp_log_msg(
      app, DK_LOG_LEVEL_INFO,
      &(messages[4]), 1
    );
    dkapp_log_msg(
      app, DK_LOG_LEVEL_PROGRESS,
      &(messages[5]), 1
    );
    dkapp_log_msg(
      app, DK_LOG_LEVEL_DEBUG,
      &(messages[6]), 1
    );
    dkapp_log_msg(
```

```
        app, DK_LOG_LEVEL_INFO,
        an_array_of_strings, 3
      );
      success = 1;
      dkapp_close(app); app = NULL;
    }
    success = (success ? 0 : 1);
    exit(success);
    return success;
  }
```

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

```
stc h04msg.str .
```

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

```
./hello04
```

9.2.6 Retrieving command line arguments

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

```
#include <dkapp.h>

#include <stdio.h>

#if DK_HAVE_STDLIB_H
#include <stdlib.h>
#endif

dk_app_t *app = NULL;

static int success = 0;

int main(int argc, char *argv[])
{
  int my_argc, i;
  char **my_argv, **lfdptr;
  app = dkapp_open(argc, argv);
  if(app) {
    success = 1;
    my_argc = dkapp_get_argc(app);
    my_argv = dkapp_get_argv(app);
    lfdptr = my_argv;
    i = 0;
    while(i < my_argc) {
      if(*lfdptr) {
        printf("%5d \"%s\"\n", i, *lfdptr);
      }
      i++; lfdptr++;
    }
    dkapp_close(app); app = NULL;
  }
  success = (success ? 0 : 1);
  exit(success);
  return success;
}
```

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.
After closing the application using `dkapp_close()` the pointer returned by `dkapp_get_argv()` can no longer be used.

9.3 Memory allocation

In lesson 9.4 on page 120 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:

```
#include <dksto.h>
#include <dkmem.h>

typedef struct {
  unsigned long customer_id;
  double        how_much;
} bank_account;

bank_account *
new_bank_account(unsigned long cid)
{
  bank_account *back = NULL;
  back = dk_new(bank_account,1);
  if(back) {
    back->customer_id = cid;
    back->how_much = 0.0;
  }
  return back;
}

void
delete_bank_account(bank_account *baptr)
{
  if(baptr) {
    dk_delete(baptr);
  }
}
```

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.

9.4 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:

```
#define COMPARE_BY_UID      0
#define COMPARE_BY_MONEY    1
#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.

```
int main(int argc, char *argv[])
{
  dk_storage_t *st_cid, *st_val;
  dk_storage_iterator_t *it_cid, *it_val;

  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 */
  }
  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;
}
```

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 {
    delete_bank_account(baptr1);
  }
}
```

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:

```
dksto_remove(st_val,baptr1);
baptr1->how_much = value;
if(!dksto_add(st_val,baptr1)) {
  dksto_remove(st_cid,baptr1);
  delete_bank_account(baptr1);
}
```

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:

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

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

9.5 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.

9.5.1 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`.

```
static
int
copy_streams DK_P2(
  k_stream_t *, dst, dk_stream_t *, src
)
{
  int back;
  char buffer[MY_BUFFERSIZE];
  int  can_continue;
  size_t bufused, bufwritten;
  can_continue = back = 1;
  while(can_continue) {
    bufused =
    dkstream_read(src,buffer,sizeof(buffer));
    if(bufused) {
      bufwritten =
      dkstream_write(dst,buffer,bufused);
      if(bufwritten != bufused) {
        can_continue = 0;
        back = 0;
      }
    } else {
      can_continue = 0;
    }
  }
```

```
  return back;
}
```

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

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

```
int main(int argc, char *argv[])
{
  int exval = 0;
  int inputtype, outputtype;
  dk_stream_t *instr, *outstr;
  char *p1, *p2;
  if(argc == 3) {
    inputtype = outputtype = 0;
    p1 = dksf_get_file_type_dot(argv[1]);
    p2 = dksf_get_file_type_dot(argv[2]);
    inputtype = get_type(p1);
    outputtype = get_type(p2);
    instr = outstr = NULL;
    switch(inputtype) {
      case 1:
        instr =
        dkstream_opengz(argv[1], "rb", 0, NULL);
      break;
      case 2:
        instr =
        dkstream_openbz2(argv[1], "rb", 0, NULL);
      break;
      default :
        instr =
        dkstream_openfile(argv[1], "rb", 0, NULL);
      break;
    }
    switch(outputtype) {
      case 1:
        outstr =
        dkstream_opengz(argv[2], "wb", 0, NULL);
      break;
      case 2:
        outstr =
        dkstream_openbz2(argv[2], "wb", 0, NULL);
      break;
      default:
        outstr =
        dkstream_openfile(argv[2], "wb", 0, NULL);
```

```
      break;
    }
    if(instr && outstr) {
      exval = copy_streams(outstr, instr);
    }
    if(outstr) {
      dkstream_close(outstr); outstr = NULL;
    }
    if(instr)  {
      dkstream_close(instr);  instr  = NULL;
    }
  }
  exval = (exval ? 0 : 1);
  exit(exval); return exval;
}
```

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 memory 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.

9.5.2 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.

9.5.3 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.

9.5.4 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.

10 Appendix

10.1 Preferences Overview

10.1.1 General preferences

Table 15: Preferences

Preference	Value
/ui/lang	language code for language_region.encoding
/ui/lang/env	boolean, indicates whether the LANG environment variable overwrites /ui/lang
/dir/app	directory name for application base directory
/dir/shared	directory name for shared directory
/dir/tmp	directory name of temporary directory
/log/file/name	file name for log file
/log/file/level	minimum message priority for logging to file
/log/file/keep	priority necessary to keep log file when program exits
/log/file/codepage	file name of codepage file to use when logging to file
/log/file/time	boolean, indicates whether to include date and time in log messages printed to file
/log/file/split	boolean, indicates whether to break log file lines after date and time
/log/stdout/level	minimum message priority for logging to standard output
/log/stdout/time	boolean, indicates whether to include date and time in log messages printed to standard output
/log/stdout/codepage	file name of codepage file to use when logging to standard output
/log/stdout/split	boolean, indicates whether to break log lines after date and time when logging to standard output
/log/stderr/level	minimum message priority for logging to standard error stream

. . . to be continued

Continuation	
`/log/stderr/codepage`	file name of codepage file to use when logging to standard error stream
`/log/stderr/time`	boolean, indicates whether to include date and time when logging to standard error stream
`/log/stderr/split`	boolean, indicates whether to break log lines after date and time when logging to standard error stream

`/log/stdout/ide`
`/log/stderr/ide`
`/log/file/ide`
	string, allows an application to behave like a compiler if it knows about a source file name and line number.
	Choices are as follows:
	• `none`
	Normal behaviour.
	• `g$cc`
	Simulate Gnu C compiler.
	• `ms$vc`
	Simulate MS Visual C++ 5.0 compiler.
	• `w$orkshop`
	Simulate Sun Workshop C compiler.
	• `t$asm`
	Simulate Borland C 3.1's Turbo Assembler.
	You can specify either `gcc` or simply the abbreviation `g`, abbreviation is allowed after the dollar sign.

`/storage/trees`	boolean, indicates whether tree structures are allowed for sorted storage (do not set this)

10.1.2 Preferences for security checks

These preferences – if used – decrease security and can be set on command line only.
Be sure to know what you are doing.

Table 16: Preferences for security checks

Preference	Value
`/sec/ign/link-owner`	Boolean, skips the check whether the owner of a symbolic link is the owner of the target file.
`/sec/ign/dir-group-writable`	Boolean, skips the check whether the symbolic link resides in a group writable directory.
`/sec/ign/dir-world-writable`	Boolean, skips the check whether the symbolic link resides in a group- or world-writable directory.

10.2 Macros in preferences

The following macros can be used to define `/log/file/name`, `/dir/app`, `/dir/shared` and `/dir/tmp`.
Example:

```
"/dir/app"="/usr/local/lib/$(app.name)"
```

The translation is done by the
`int dkapp_transform_string(dk_app_t *a, char *d, size_t sz, char *s)`
function.

Table 17: Preferences

Macro	Replacement text
`app.name`	the name of the current application
`user.name`	the users login name
`user.home`	the users home directory
`user.uid`	the users UID
`user.gid`	the users GID
`user.euid`	the users effective UID
`user.egid`	the users effective GID
`host.name`	the host name (short)
`host.domain`	the DNS domain the host belongs to
`app.dir`	the application base directory (can not be used to define `/dir/app`)
`shared.dir`	the shared directory (can not be used to define `/dir/shared`)
`temp.dir`	the temporary directory (can not be used to define `/dir/tmp`)
`process.pid`	the PID of the current process
`process.ppid`	the PID of the parent process
`process.pgid`	the process group ID

10.3 Search order for the dkapp_find_file() function

In this section we use the following placeholders:

- *prefix*
  is the installation prefix (the directory where the software on your system resides).
  On U*x systems this is mostly /usr/local, on W* systems C:\Programs is oftenly used.

- *appname*
  is the name of the application without the leading directory and a trailing suffix (i.e. `.exe`).

- *groupname*
  is the name of the software package the application belongs to. I.e. the programs `klpr`, `klpq`, `klprm`, `klpc` and `snmpyalc` belong to the group `yanolc`.

- *application-directory*
  is a directory containing resources for a specific application. This is set via the `/dir/app` preference. If this preference is not defined the directory defaults to *prefix*/lib/*appname*.

- *shared-directory*
  is a directory containing resources used by several applications. This is set via the `/dir/shared` preference. If this preference is not defined the directory defaults to *prefix*/lib.

- *%WINDIR%*
  is your Windows directory which is found by the `GetWindowsDirectoryA()` function (on W* systems only).

- *sysconfdir*
  is the directory containing configuration files on your system. This is configured via the `-sysconfdir=` option when running `./configure` on U*x systems and defaults to *prefix*/etc or `/etc`.

- *language*, *region* and *encoding*
  are the preferred languagem, region and encoding as specified by the `/ui/l` preference or the `LANG` environment variable. A value of `en_US.UTF-8` sets the language to `en`, the region to `us` and the encoding to `utf-8`.

The `dkapp_find_file()` function searches for

filename
filename`.gz` when compiled with gzip support,
filename`.bz2` when compiled with bzip2 support

in the following directories:

current directory
application-directory/*language*/*region*/*encoding*
application-directory/*language*/*region*
application-directory/*language*/*encoding*
application-directory/*language*
application-directory/*region*/*encoding*
application-directory/*region*
application-directory/*encoding*
application-directory
shared-directory/*groupname*/*language*/*region*/*encoding*
shared-directory/*groupname*/*language*/*region*
shared-directory/*groupname*/*language*/*encoding*
shared-directory/*groupname*/*language*
shared-directory/*groupname*/*region*/*encoding*
shared-directory/*groupname*/*region*
shared-directory/*groupname*/*encoding*
shared-directory/*groupname*
shared-directory/*language*/*region*/*encoding*
shared-directory/*language*/*region*
shared-directory/*language*/*encoding*
shared-directory/*language*
shared-directory/*region*/*encoding*
shared-directory/*region*
shared-directory/*encoding*
shared-directory
%WINDIR%/*appname*/*language*/*region*/*encoding*
%WINDIR%/*appname*/*language*/*region*
%WINDIR%/*appname*/*language*/*encoding*
%WINDIR%/*appname*/*language*
%WINDIR%/*appname*/*region*/*encoding*
%WINDIR%/*appname*/*region*
%WINDIR%/*appname*/*encoding*
%WINDIR%/*appname*
%WINDIR%/*groupname*/*language*/*region*/*encoding*
%WINDIR%/*groupname*/*language*/*region*

%WINDIR%/groupname/language/encoding
%WINDIR%/groupname/language
%WINDIR%/groupname/region/encoding
%WINDIR%/groupname/region
%WINDIR%/groupname/encoding
%WINDIR%/groupname
%WINDIR%/language/region/encoding
%WINDIR%/language/region
%WINDIR%/language/encoding
%WINDIR%/language
%WINDIR%/region/encoding
%WINDIR%/region
%WINDIR%/encoding
%WINDIR%
sysconfdir/appname/language/region/encoding
sysconfdir/appname/language/region
sysconfdir/appname/language/encoding
sysconfdir/appname/language
sysconfdir/appname/region/encoding
sysconfdir/appname/region
sysconfdir/appname/encoding
sysconfdir/appname
sysconfdir/groupname/language/region/encoding
sysconfdir/groupname/language/region
sysconfdir/groupname/language/encoding
sysconfdir/groupname/language
sysconfdir/groupname/region/encoding
sysconfdir/groupname/region
sysconfdir/groupname/encoding
sysconfdir/groupname
sysconfdir/language/region/encoding
sysconfdir/language/region
sysconfdir/language/encoding
sysconfdir/language
sysconfdir/region/encoding
sysconfdir/region
sysconfdir/encoding
sysconfdir

10.4 Search order for the dkapp_find_cfg() function

The `dkapp_find_cfg()` function searches for

filename
filename`.gz` when compiled with gzip support,
filename`.bz2` when compiled with bzip2 support

in the following directories:
%WINDIR%/appname/language/region/encoding
%WINDIR%/appname/language/region
%WINDIR%/appname/language/encoding
%WINDIR%/appname/language
%WINDIR%/appname/region/encoding
%WINDIR%/appname/region
%WINDIR%/appname/encoding
%WINDIR%/appname
%WINDIR%/groupname/language/region/encoding
%WINDIR%/groupname/language/region
%WINDIR%/groupname/language/encoding
%WINDIR%/groupname/language
%WINDIR%/groupname/region/encoding
%WINDIR%/groupname/region
%WINDIR%/groupname/encoding
%WINDIR%/groupname
%WINDIR%/language/region/encoding
%WINDIR%/language/region
%WINDIR%/language/encoding
%WINDIR%/language
%WINDIR%/region/encoding
%WINDIR%/region
%WINDIR%/encoding
%WINDIR%
sysconfdir/appname/language/region/encoding
sysconfdir/appname/language/region
sysconfdir/appname/language/encoding
sysconfdir/appname/language
sysconfdir/appname/region/encoding
sysconfdir/appname/region
sysconfdir/appname/encoding
sysconfdir/appname
sysconfdir/groupname/language/region/encoding

sysconfdir/*groupname*/*language*/*region*
sysconfdir/*groupname*/*language*/*encoding*
sysconfdir/*groupname*/*language*
sysconfdir/*groupname*/*region*/*encoding*
sysconfdir/*groupname*/*region*
sysconfdir/*groupname*/*encoding*
sysconfdir/*groupname*
sysconfdir/*language*/*region*/*encoding*
sysconfdir/*language*/*region*
sysconfdir/*language*/*encoding*
sysconfdir/*language*
sysconfdir/*region*/*encoding*
sysconfdir/*region*
sysconfdir/*encoding*
sysconfdir