

# LibDsk v1.5.9-pre

ohn Elliott

## Abstract

LibDsk is a library intended to give transparent access to floppy drives and to the “disc image files” used by emulators to represent floppy disks.

This library is free software, released under the GNU Library GPL. See COPYING for details.

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	About this document . . . . .	6
1.2	About LibDsk . . . . .	6
1.3	What's new? . . . . .	7
1.4	Terms and definitions . . . . .	8
<b>2</b>	<b>Supported file formats</b>	<b>8</b>
<b>3</b>	<b>Architecture</b>	<b>9</b>
3.1	Logical and physical sectors . . . . .	9
3.1.1	DSK_GEOMETRY in detail . . . . .	10
<b>4</b>	<b>LibDsk Function Reference</b>	<b>11</b>
4.1	dsk_open: Open an existing disc image . . . . .	11
4.2	dsk_creat: Create a new disc image . . . . .	12
4.3	dsk_close: Close a drive or disc image . . . . .	12
4.4	dsk_dirty: Read the dirty flag . . . . .	21
4.5	dsk_pread, dsk_lread : Read a sector . . . . .	12
4.6	dsk_pwrite, dsk_lwrite: Write a sector . . . . .	13
4.7	dsk_pcheck, dsk_lcheck: Verify sectors on disc against memory . . . . .	13
4.8	dsk_pformat, dsk_lformat: Format a disc track . . . . .	13
4.9	dsk_apform, dsk_alform: Automatic format . . . . .	14
4.10	dsk_psecid, dsk_lsecid: Read a sector ID. . . . .	14
4.11	dsk_ptrackids, dsk_ltrackids: Identify sectors on a track. . . . .	15
4.12	dsk_rtread: Reserved. . . . .	51
4.13	dsk_xread, dsk_xwrite: Low-level reading and writing . . . . .	15
4.13.1	dsk_xread(), dsk_xwrite(): Deleted data . . . . .	15
4.14	dsk_lthead, dsk_ptread, dsk_xtread . . . . .	16
4.15	dsk_lseek, dsk_pseek . . . . .	16
4.16	dsk_drive_status . . . . .	17
4.17	dsk_dirty: Has drive been written to? . . . . .	17
4.18	dsk_getgeom: Guess disc geometry . . . . .	17
4.19	dg_*geom : Initialise disc geometry from boot sector . . . . .	17
4.20	dg_stdformat : Initialise disc geometry from a standard LibDsk format. . . . .	18
4.21	dsk_*_forcehead: Override disc head . . . . .	19
4.22	dsk_*_option: Set/get driver option . . . . .	19
4.22.1	Filesystem driver options . . . . .	20
4.23	dsk_option_enum: Get list of driver options . . . . .	21
4.24	dsk_*_comment: Set comment for disc image . . . . .	21
4.25	dsk_type_enum . . . . .	21
4.26	dsk_comp_enum . . . . .	21
4.27	dsk_drvname, dsk_drvdesc . . . . .	22
4.28	dsk_compname, dsk_compdesc . . . . .	22
4.29	dg_ps2ls, dg_ls2ps, dg_pt2lt, dg_lt2pt . . . . .	22
4.30	dsk_strerror: Convert error code to string . . . . .	22
4.31	dsk_reportfunc_set / dsk_reportfunc_get . . . . .	23
4.32	dsk_set_retry / dsk_get_retry . . . . .	23
4.33	dsk_get_psh . . . . .	23

4.34	dsk_copy: Copy an entire disk image . . . . .	23
4.35	Structure: DSK_FORMAT . . . . .	24
4.36	LibDsk errors . . . . .	24
4.37	Miscellaneous . . . . .	25
5	Initialisation files . . . . .	25
5.1	libdiskrc format . . . . .	25
5.1.1	libdiskrc example . . . . .	26
5.2	Locating libdiskrc . . . . .	26
5.2.1	UNIX . . . . .	26
5.2.2	Win32 . . . . .	27
5.2.3	Win16 . . . . .	27
5.2.4	DOS . . . . .	27
6	Reverse CP/M-FS (rcpmfs) backend . . . . .	27
6.1	In Use . . . . .	27
6.2	rcpmfs initialisation file . . . . .	28
6.3	Bugs . . . . .	29
7	LibDsk under Windows . . . . .	29
7.1	Windows 3.x . . . . .	30
7.2	Windows 4.x (95, 98 and ME) . . . . .	30
7.3	Windows NT (NT 3.x, NT 4.x, 2000, XP) without ntwdm driver . . . . .	30
7.4	Windows 2000 and XP with ntwdm driver . . . . .	30
7.5	General comments on programming floppy access for Windows . . . . .	30
7.5.1	The Win16 driver. . . . .	30
7.5.2	The Win32c driver. . . . .	30
7.5.3	The Win32 driver. . . . .	31
7.5.4	The ntwdm driver. . . . .	31
7.5.5	Other floppy APIs . . . . .	31
7.6	LDSEVER . . . . .	31
7.6.1	Compiling LDSEVER . . . . .	31
7.6.2	Using LDSEVER . . . . .	31
7.6.3	Important Security Warning . . . . .	31
7.7	LibDsk and COM . . . . .	32
7.7.1	General points . . . . .	32
7.7.2	Library . . . . .	32
7.7.3	Geometry . . . . .	32
7.7.4	Disk . . . . .	33
7.7.5	IReporter . . . . .	34
8	LibDsk RPC system . . . . .	34
8.1	The 'serial' driver . . . . .	35
8.1.1	Servers for the serial driver . . . . .	35
8.2	The 'fork' driver . . . . .	36

<b>9</b>	<b>Writing new drivers</b>	<b>36</b>
9.1	The driver header . . . . .	36
9.2	The driver source file . . . . .	37
9.3	Driver functions . . . . .	38
9.3.1	dc_open . . . . .	38
9.3.2	dc_creat . . . . .	38
9.3.3	dc_close . . . . .	38
9.3.4	dc_read . . . . .	39
9.3.5	dc_write . . . . .	39
9.3.6	dc_format . . . . .	39
9.3.7	dc_getgeom . . . . .	39
9.3.8	dc_secid . . . . .	40
9.3.9	dc_xseek . . . . .	40
9.3.10	dc_xread, dc_xwrite . . . . .	40
9.3.11	dc_status . . . . .	40
9.3.12	dc_tread . . . . .	40
9.3.13	dc_xtread . . . . .	41
9.3.14	dc_option_enum . . . . .	41
9.3.15	dc_option_set, dc_option_get . . . . .	41
9.3.16	dc_trackids . . . . .	42
9.3.17	dc_rtread . . . . .	42
9.3.18	dc_to_ldbs . . . . .	42
9.3.19	dc_from_ldbs . . . . .	42
<b>10</b>	<b>Writing new drivers (derived from LDBS)</b>	<b>42</b>
10.1	The driver header . . . . .	42
10.2	The driver source file . . . . .	43
10.3	Driver functions . . . . .	44
10.3.1	dc_open . . . . .	44
10.3.2	dc_creat . . . . .	45
10.3.3	dc_close . . . . .	45
<b>11</b>	<b>Adding new compression methods</b>	<b>46</b>
11.1	Driver header . . . . .	46
11.2	Driver implementation . . . . .	46
11.3	Compression functions . . . . .	47
11.3.1	cc_open . . . . .	47
11.3.2	cc_creat . . . . .	48
11.3.3	cc_commit . . . . .	48
11.3.4	cc_abort . . . . .	48
<b>12</b>	<b>Adding new remote transports.</b>	<b>48</b>
12.1	Driver header . . . . .	48
12.2	Driver implementation . . . . .	48
12.3	Remote communication functions . . . . .	49
12.3.1	rc_open . . . . .	49
12.3.2	rc_close . . . . .	50
12.3.3	rc_call . . . . .	50

<b>A</b>	<b>The CopyQM File Format</b>	<b>50</b>
<b>A.1</b>	<b>Introduction . . . . .</b>	<b>50</b>
<b>A.2</b>	<b>Header . . . . .</b>	<b>50</b>
<b>A.3</b>	<b>CRC . . . . .</b>	<b>51</b>
<b>A.4</b>	<b>Image comment . . . . .</b>	<b>51</b>
<b>A.5</b>	<b>Image data . . . . .</b>	<b>51</b>
<b>B</b>	<b>DQK Files</b>	<b>52</b>
<b>C</b>	<b>LibDsk with cpmtools</b>	<b>52</b>
<b>D</b>	<b>DSK / EDSK recording mode extension</b>	<b>53</b>

# 1 Introduction

## 1.1 About this document

This document only covers LibDsk the library itself. For information on the example utilities supplied with LibDsk (apriboot, dskformsktrans, dskid, dskdump, dskscan, dskutil and md3serial) see their respective pages.

## 1.2 About LibDsk

LibDsk is a library for accessing floppy drives and disc images transparently. It currently supports the following disc image formats:

Raw “dd if=foo of=bar” images

Raw images in logical filesystem order

CPCEMU-format .DSK images (normal and extended)

MYZ80-format hard drive images

CFI-format disc images, as produced by FDCOPY.COM under DOS and used to distribute some Amstrad system discs

ApriDisk-format disc images, used by the utility of the same name under DOS.

NanoWasp-format disc images, used by the eponymous emulator

IMD-format disc images, as produced by Dave Dunfield’s InaDisk utility.

Yaze ’ydisk’ disc images, created by the ’yaze’ and ’yaze-ag’ulators.

V3-format disc images, used in TRS-80 emulation.

Compaq Quick Release Sector Transfer (QRST), used for the computers’ BIOS setup floppies.

Disc images created by the Sydex imaging programs Teledisk and CopyQM (Not supported in the 16-bit Windows version).

SAP disc images used in emulation of Thomson computers.

The floppy drive under Linux

The floppy drive under Windows. Windows support is a complicated subject - see section 7 below.

The floppy drive (and hard drive partitions) under DOS.

LDBS: A disc image format still under development. See LDBS.html for more information.

LibDsk also supports compressed disc images in the following formats:

Squeeze (Huffman coded)

GZip (Deflate )

BZip2 (Burrows-Wheeler support is read-only)

TeleDisk ’advanced’ compression (LZH support is read-only and confined to TeleDisk disk images)

### 1.3 What's new?

**Important note:** If you have coded against a version of LibDsk prior to 1.5.3, be in mind that 1.5.3 is a substantial rewrite. The API for your programs remains the same, but there are differences in implementation that may trip you up. In particular, if you are writing to a disc image file it is very important to check the result of `dsk_close()` there are a lot of drivers which make their changes in memory and don't try to commit them until `dsk_close()` is called.

For full details, see the file `ChangeLog`.

`dskform` can now format disc images with a blank PCDOS or `ApriMSDOS` filesystem.

Added a new 'sap' driver for SAP-format disc images (thanks to `Emulix75` for information).

When converting to Teledisk TD0 format, attempt to come up with a plausible value for the drive type (3.5" / 5.25").

The geometry probe has been amended to try and get the correct data rate on disc images that don't include this information as metadata.

The 'dsk' / 'edsk' driver has been rewritten to support the extensions described at <http://simonowen.com/misc/extextdsk.txt>

The LDBS file format has been changed to support the above extensions. LDBS files created by LibDsk 1.5.3 and earlier can still be used, though don't support the extensions.

Added a new 'qrst' driver for Compaq QRST-format disc image

Various bugfixes in the IMD driver.

Added a new 'ldbs' driver for LDBS-format disc images.

`CopyQM` and Teledisk are no longer supported in 16-bit Windows builds, because 16-bit Windows does not provide the `scanf()` function in DLLs.

Added a new 'complement' option to the drive geometry, allowing for disc image formats where the bytes are stored complemented.

Added a new 'jv3' driver for V3-format disc images.

A bugfix to the automatic geometry probe in the 'imd' driver. HD discs were not being correctly detected.

Added a new 'imd' driver for IMD-format disc images.

A new `SIDES_EXTSURFACE` geometry, for disc images where the sector numbers on side 1 follow on from side 0.

TeleDisk images with 'advanced' (LZH) compression are now supported.

Added a new 'ydisk' driver for YAZE ydisk-format disc images.

Some disc image files include filesystem information as part of the disc image metadata. `dskid` and `dsktrans` now display and copy this information.

Should now compile out of the box on FreeBSD.

A bugfix to the rcpmfs driver should allow it to simulate a CP2 filesystem as well as CPM 3.

## 1.4 Terms and definitions

In this document, I use the word **CYLINDER** to refer to a position on a floppy disc, and **TRACK** to refer to the data within a cylinder on one side of the disc. On a single-sided disc, these are the same for a double-sided disc, there are as many tracks as cylinders.

## 2 Supported file formats

The following disc image file formats are supported by LibDsk

“dsk” : Disc image in the DSK format used by CPCEMU. The format of a **DSK** file is described in the CPCEMU documentation.

“edsk” : Disc image in the extended CPCEMU DSK format.

“raw” : Raw disc image - as produced by “`dd if=q6\1q0 of=msd6”`. On systems other than Linux, DOS or Windows, this is also used to ~~acc~~ the host system’s floppy drive.

“rawoo” : Raw disc image, ordered so that all the tracks on side 0 come ~~first~~ then all the tracks on side 1.

“rawob” : Raw disc image, ordered so that all the tracks on side 0 come ~~first~~ then all the tracks on side 1 in reverse order.

“logical” : Raw disc image in logical filesystem order. Early versions of LibDsk could generate such images (for example, by using the now-depre~~ted~~ `-logical` option to `dsktrans`) but couldn’t then write them back or use ~~them~~ in emulators.

“floppy” : Host system’s floppy drive (under Linux, DOS or Windows).

“int25” : Hard drive partition under DOS. Also used for the floppy ~~drive~~ Apricot PCs.

“ntwdm” : Enhanced floppy support under Windows 2000 and XP, using ~~additional~~ kernel-mode driver.

“myz80” : MYZ80 hard drive image, which is ~~is~~ nearly the same as “raw” but has a 256 byte header.

“cfi” : Compressed floppy image, as produced by FDCOPY.COM under ~~SD~~ts format is described in `cfi.html`.

“imd” : Disc images created by Dave Dunfield’s ImageDisk utility.

“jv3” : Disc images used by eff Vavasour’s TRS-80 emulators.

“qm” : Disc images created by Sydex’s CopyQM.

“tele” : Disc images created by Sydex’s TeleDisk.

“nanowasp” : Disc image in the 400k Microbee format used by the NanoWasplator. This is similar to “raw”, but the tracks are stored in different order. LibDsk also applies a sector skew so that the sectors are read/written in the logical order. Strictly speaking, it should not do this (when libdsk is used with cpmtools, cpmtools is the one that does the skewing) but cpmtools cannot handle the skewing scheme used by the Microbee format.

“apridisk”: Disc image in the format used by the ApriDisk utility. The format is described in apridisk.html.

“rcpmfs”: Reverse CP/M filesystem. A directory is made to appear as a CPdisk. This is a complex system and should be approached with caution.

“remote”: Remote LibDsk server, most likely at the other end of a serial line.

“ydisk”: Disc image format used by the yaze and yaze-ag CP/M emulators

“qrst”: Compaq Quick Release Sector Transfer.

“ldbs”: LibDsk Block Store.

### 3 Architecture

LibDsk is composed of a fixed core (files named `libdsk.*`) and a number of drivers (files named `q2k-*.c`). When you open an image or a drive (using `q2k-open()` or `q2k-clget()`) then a driver is chosen. This driver is then used until it is closed (`q2k-close()`).

Each driver is identified by a name. To get a list of available drivers, use `q2k-libgetnames()`. To get the driver that is being used by an open DSK image, use `q2k-qlname()` or `q2k-qlqname()`.

#### 3.1 Logical and physical sectors

LibDsk has two models of disc geometry. One is as a linear array of “logical” sectors - for example, a 720k floppy appears as 1440 512-byte sectors numbered 0 to 1439. The other locates each sector using a (Cylinder, Head, Sector) triple - so on the 720k floppy described earlier, sectors would run from (0,0,1) to (79,1,1439).

Internally, all LibDsk drivers are written to use the Cylinder/Head/Sector model. For those calls which take parameters in logical sectors, LibDsk uses the information in a `D2K_GEOMETRY` structure to convert to C/H/S. `D2K_GEOMETRY` also contains information such as the sector size and data rate used to access the disc.

Those functions which deal with whole tracks (such as the command to format a track) use logical tracks and (cylinder,head) pairs instead. To initialise a `D2K_GEOMETRY` structure, either:

call `q2k-getgeom()` to try and detect it from the disc or

call `q2k-stdgeom()` to select one of the “standard” formats that LibDsk knows about or

call `q8_q02860w()` / `q8_cbw90860w()` / `q8_bcM860w()` / `q8_gbl1860w()` to initialise it from a copy of a DOS / CP/M86 / PCW / Apricot boot sector or

Set all the members manually.

### 3.1.1 DSK\_GEOMETRY in detail

`q8_q02860w()`

`q8_q02860w()` /\* This describes the logical sequence of tracks on the disc - the order in which their host system reads them. It will only be used if `q8_m69q2` is greater than 1 (otherwise all the methods are equivalent) and you are using functions that take logical sectors or tracks as parameters. It will be one of:

`SIDES_VFI` The tracks are ordered Cylinder 0 Head 0 C0H1 C1H0 C1H1 C2H0 C2H1 etc. This layout is used by most PC-hosted operating systems, including DOS and Linux. Amstrad's 8-bit operating systems also use this ordering.

`SIDES_QCIBVCK` The tracks go out to the edge on Head 0, and then back in on Head 1 (so Cylinder 0 Head 0 is the first track, while Cylinder 1 Head 1 is the last). This layout is used by Freek Heite's 144FEAT disk (for CP/M-86 on the PC) but I have not seen it elsewhere.

`SIDES_QCLOCI` The tracks go out to the edge on Head 0, then out again on Head 1 (so the order goes C(last)H0, C0H1, C1H1, ..., C(last)H1). This ordering is used by Acorn-format discs.

`SIDES_EXI2QBEVCE` The tracks are arranged in the same way as `SIDES_ALT`, but if the sectors on side 0 are numbered 1 - the sectors on side 1 are numbered 1 - 2\* n (for example, side 0 are numbered 1-9, and side 1 are numbered 10-18). This is a new option and should be treated with caution

\*/

`q8_qcyl1` /\* The number of cylinders this disc has. Usually 40 or 80. \*/

`q8_qh69q` /\* The number of heads (sides) the disc has. Usually 1 or 2. \*/

`q8_q26ct1` /\* The number of sectors per track. \*/

`q8_q26ct1` /\* The first physical sector number. Most systems start numbering their sectors at 1 Acorn systems start at 0, and Amstrad CPCs start at 65 or 193. \*/

`q8_q26ct1` /\* Sector size in bytes. Note that several drivers rely on this being a power of 2. \*/

`q8_q181818` /\* Data rate. This will be one of:

`BYLE_HD` High-density disc (1.4Mb or 1.2Mb)

```

    BVLLE_DD Double-density disc in 1.2Mb drive (ie, 360k disc in 1.2Mb drive)
    BVLLE_2D Double-density disc in 1.4Mb or 720k drive
    BVLLE_ED Extra-density disc (2.8Mb) */
qzkr_gsb_t qg_LM_gsb /* Read/write gap length */
qzkr_gsb_t qg_fm_gsb /* Format gap length */
int qg_fm /* This is really a dsk_recmode_t, but is declared as an int for backward
compatibility. It contains the recording mode and additional flags. To extract the
recording mode, use (dg_fm RECMODE_MASK):

    RECMODE_MFM (double density) recording mode.
    RECMODE_FM (single density) recording mode. Not all PC floppy controllers
support this mode the National Semiconductor PC87306 and Future
Domain TMC series SCSI controllers can at least read FM discs. The BBC
Micro used FM recording for its 100k and 200k DFS formats. Windows /
DOS floppy drivers do not support FM recording.

To extract the flags, use (dg_fm RECMODE_FLAGMASK). There is currently one additional flag
RECMODE_COMPLEMENT. If this flag is set, bytes written on the disc are stored complemented
(ie, XOR with 0xFF). */

int qg_mt /* Set to nonzero to disable multitrack mode. This only affects
attempts to read normal data from tracks containing deleted data (or vice versa).
*/

int qg_ns /* Set to nonzero to disable skipping deleted data when searching
for non-deleted data (or vice versa). */

```

D2K\_GEOMETRY

## 4 LibDsk Function Reference

### 4.1 dsk\_open: Open an existing disc image

```

qzkr_glr_t qzkr_obcm(D2K_PDRIVER *self, const char *filename, const char *type, const char *comment)

```

Enter with:

“self” is the address of a DSK\_PDRIVER variable (treat it as a handle to a drive / disc file). On return, the variable will be non-null (if the operation succeeded) or null (if the operation failed).

“filename” is the name of the disc image file. On DOS and Windows, “A:” and “B:” refer to the two floppy drives. On Apricot MS-DOS, “0:” and “1:” refer to the floppy drives.

“type” is NULL to detect the disc image format automatically or the name of a LibDsk driver to force that driver to be used. See `dsk_get_driver()` below.

“compress” is NULL to auto-detect compressed files, or theme of a LibDsk compression scheme. See `dsk_compress()`.

Returns: An integer which will be 0 (`D2K_EBK_OK`) if successful, or a negative integer if failed. See `dsk_err()`. The error `D2K_EBK_NOFILE` means either that no driver was able to open the disc / disc image (if “type” was NULL) or that the requested driver could not open the file (if “type” was not NULL).

Standard LibDsk drivers are listed in section 2.

Compression schemes are:

“sq” : Huffman (squeezed). The reason for the inclusion of this system is to support .DQK images (see appendix B).

“gz” : GZip (deflate). This will only be present if libdsk was built with zlib support.

“bz2” : BZip2 (Burrows-Wheeler compression). This support is currently read-only, and will only be present if LibDsk was built with bzip support.

#### 4.2 dsk\_creat: Create a new disc image

`dsk_err_t dsk_creat(D2K_DRIVE *drive, const char *filename, const char *type)`

In the case of floppy drives, this acts exactly as `dsk_open()`. For image files, the file will be deleted and recreated. Parameters and results are as `dsk_open()`, except that “type” cannot be NULL (it must specify the type of disc to be created) and if “compress” is NULL, it means that the file being created should not be compressed.

#### 4.3 dsk\_close: Close a drive or disc image

`dsk_err_t dsk_close(D2K_DRIVE *drive)`

Pass the address of an opaque pointer returned from `dsk_open()` / `dsk_creat()`. On return, the drive will have been closed and the pointer set to NULL. It is important to check the result of this function many drivers don't write their changes back until `dsk_close()` is called.

#### 4.4 dsk\_dirty: Read the dirty flag

`int dsk_dirty(D2K_DRIVE drive)`

This function returns non-zero if the disc has been modified since it was inserted into the drive, and zero if it has not been modified.

#### 4.5 dsk\_pread, dsk\_lread : Read a sector

`dsk_err_t dsk_pread(D2K_DRIVE drive, const D2K_GEOMETRY *geom, void *buf, dsk_bcyl_t cyl, dsk_sector_t sector)`  
`dsk_err_t dsk_lread(D2K_DRIVE drive, const D2K_GEOMETRY *geom, void *buf, dsk_sector_t sector)`

These functions read a single sector from the disc. There are two of them, depending on whether you are using logical or physical sector addresses.

Enter with:

“self” is a handle to an open drive / image file.

“geom” points to the geometry for the drive.

“buf” is the buffer into which data will be loaded.

“cylinder”, “head” and “sector” ( `q2k-bl69q` ) or “sector” ( `q2k-l69q` ) give the location of the sector.

**Returns:**

If successful, DSK\_ERR\_OK. Otherwise, a negative DSK\_ERR value.

If the driver cannot read sectors, DSK\_ERR\_NOTIMPL will be returned.

**4.6 dsk\_pwrite, dsk\_lwrite: Write a sector**

```
q2k-6ll- t q2k-bl69q(D2K- bDblAEb 26lf, const D2K-CEOMELbA *g6om, const void *pnt, q2k-
q2k-6ll- t q2k-l69q(D2K- bDblAEb 26lf, const D2K-CEOMELbA *g6om, const void *pnt, q2k-
```

As `dsk_pread / dsk_lread`, but write their buffers to disc rather than reading them from disc. If the driver cannot write sectors, DSK\_ERR\_NOTIMPL will be returned.

**4.7 dsk\_pcheck, dsk\_lcheck: Verify sectors on disc against memory**

```
q2k-6ll- t q2k-bcpeck(D2K- bDblAEb 26lf, const D2K-CEOMELbA *g6om, const void *pnt, q2k-
q2k-6ll- t q2k-l69q(D2K- bDblAEb 26lf, const D2K-CEOMELbA *g6om, const void *pnt, q2k-
```

As `q2k-bl69q / q2k-l69q`, but rather than reading their buffers from disc, they compare the contents of their buffers with the data already on disc. If the data match, the functions return DSK\_ERR\_OK. If there is a mismatch, they return DSK\_ERR\_MISMATCH. In case of error, other DSK\_ERR\_\* values are returned. If driver cannot read sectors, DSK\_ERR\_NOTIMPL will be returned.

**4.8 dsk\_pformat, dsk\_lformat: Format a disc track**

```
q2k-6ll- t q2k-bf0lmsr(D2K- bDblAEb 26lf, D2K-CEOMELbA *g6om, q2k-bcyl- t cylndqr, q2k-b
q2k-6ll- t q2k-lf0lmsr(D2K- bDblAEb 26lf, D2K-CEOMELbA *g6om, q2k-ltrck- t trck, const D
```

Enter with:

“self” is a handle to an open drive / image file.

“geom” points to the geometry for the drive. The format may modify this if (for example) it’s asked to format track 41 of a 40-track drive

“cylinder” / “head” ( `q2k-bf0lmsr` ) or “track” ( `q2k-lf0lmsr` ) give the location of the track to format.

“format” should be an array of ( `disk_geom_sector` ) `DISK_FORMAT` structures. These structures must contain sector headers for the track being formatted. For example, to format the first track of a 720k disc, you would pass in an array of 9 such structures: 0, 0, 1, 512, 0, 0, 2, 512, ..., 0, 0, 9, 512

“filler” should be the filler byte to use. Currently the Win32 driver ignores this parameter. If the driver cannot format tracks, `DISK_ERR_NOTIMPL` will be returned.

Note that when formatting a `.DISK` file that has more than one head, you must format cylinder 0 for each head before formatting other cylinders.

#### 4.9 `dsk_apform`, `dsk_alform`: Automatic format

`disk_ioctl(disk_device_t dev, const DISK_GEOMETRY *geom, disk_ioctl_t cmd, void *arg)`  
`disk_ioctl(disk_device_t dev, const DISK_GEOMETRY *geom, disk_ioctl_t cmd, void *arg)`

These function calls behave as `dsk_bformat()` and `dsk_lformat()` above, except that the sector headers are automatically generated. Takes time and trouble setting up sector headers on discs with standard layouts such as DOS, or Linux floppies. If the driver cannot format tracks, `DISK_ERR_NOTIMPL` will be returned.

#### 4.10 `dsk_psectid`, `dsk_lsectid`: Read a sector ID.

`disk_ioctl(disk_device_t dev, const DISK_GEOMETRY *geom, disk_ioctl_t cmd, void *arg)`  
`disk_ioctl(disk_device_t dev, const DISK_GEOMETRY *geom, disk_ioctl_t cmd, void *arg)`

Read a sector ID from the given track. This can be used to probe discs with oddly-numbered sectors (eg, numbered 65-74). Enter with:

“self” is a handle to an open drive / image file.

“geom” points to the geometry for the drive.

“cylinder” / “head” ( `disk_sector` ) or “track” ( `disk_sector` ) give the location of the track to read the sector from.

“result” points to an uninitialised `DISK_SECTOR` structure.

On return:

If successful, the buffer at “result” will be initialised with the sector header found, and `DISK_ERR_OK` will be returned.

If the driver cannot provide this functionality (for example, the Win32 driver under NT), `DISK_ERR_NOTIMPL` will be returned.

Note that the DOS, Win16 and Win32 (under Win9x) drivers implement a limited version of this call, which will work on normal DOS / CP/M86 / PCWds and CPC discs. However it will not be usable for other purposes.

#### 4.11 dsk\_ptrackids, dsk\_ltrackids: Identify sectors on track.

```
dsk_ptrackids(DISK_FORMAT *format, int track, int start, int end, int *ids)
dsk_ltrackids(DISK_FORMAT *format, int track, int start, int end, int *ids)
```

These functions are intended to read all the sector IDs from track, in order, and (preferably) starting at the index hole. If they succeed, start will point at an array of DISK\_FORMAT structures describing the sectors found. This array will have been allocated with dsk\_malloc() and should be freed with dsk\_free().

#### 4.12 dsk\_rtread: Reserved.

```
dsk_rtread(DISK_FORMAT *format, int track, int start, int end, int *ids)
```

This function is reserved for future expansion. The intent is to use it for diagnostic read commands (such as reading the raw bits from a track). Currently it returns DISK\_ERR\_NOTIMPL.

#### 4.13 dsk\_xread, dsk\_xwrite: Low-level reading and writing

```
dsk_xread(DISK_FORMAT *format, int track, int start, int end, int *ids)
dsk_xwrite(DISK_FORMAT *format, int track, int start, int end, int *ids)
```

dsk\_xread() and dsk\_xwrite() are extended versions of dsk\_read() and dsk\_write(). They allow the caller to read/write sectors whose sector ID differs from the physical location of the sector, or to read/write deleted data.. The “finder” and “head” arguments specify where to look the “cyl\_expected” and “head\_expected” are the values to search for in the sector header.

These functions are only supported by the CPCEMU driver, the x86 floppy driver and the NTWDM floppy driver. Other drivers will return DISK\_ERR\_NOTIMPL. Unless you are emulating a floppy controller, or you need to read disks that contain deleted data or misnumbered sectors, it should not be necessary to use these functions.

##### 4.13.1 dsk\_xread(), dsk\_xwrite(): Deleted data

The “deleted” argument is used if you want to read or write sectors that have been marked as deleted. In dsk\_xwrite(), this is a simple value pass 0 to write normal data, or 1 to write deleted data. In dsk\_xread(), pass the address of an integer containing 0 (read normal data) or 1 (read deleted data). On return, the integer will contain:

If the requested data type was read: 0

If the other data type was read: 1

If the command failed: Value is meaningless.

Passing NULL acts the same as passing a pointer to 0.

The opposite type of data will only be read if you set format->deleted to nonzero. Some examples:

geom-dg_noskip	deleted	Data on disc	Results	*deleted becomes
0	- 0	Normal	DSK_ERR_OK	0
0	- 0	Deleted	DSK_ERR_NODATA	??
0	- 1	Deleted	DSK_ERR_NODATA	??
1	- 0	Normal	DSK_ERR_OK	0
1	- 0	Deleted	DSK_ERR_OK	1
1	- 1	Normal	DSK_ERR_OK	1
1	- 1	Deleted	DSK_ERR_OK	0

#### 4.14 dsk\_lread, dsk\_pread, dsk\_xread

```

dsk_lread(DISK_DEVICE dev, const DISK_GEOMETRY *geom, void *buf, dsk_track_t track)
dsk_pread(DISK_DEVICE dev, const DISK_GEOMETRY *geom, void *buf, dsk_cyl_t cyl)
dsk_xread(DISK_DEVICE dev, const DISK_GEOMETRY *geom, void *buf, dsk_cyl_t cyl)

```

These functions read a track from the disc, using the FDC's "READ TRACK" command. There are three of them - logical, physical and extended physical.

If the driver does not support this functionality, LibDisk will attempt to simulate it using multiple sector reads.

Enter with:

"self" is a handle to an open drive / image file.

"geom" points to the geometry for the drive.

"buf" is the buffer into which data will be loaded.

"cylinder" and "head" ( `dsk_cyl_t` , `dsk_head_t` ) or "track" ( `dsk_track_t` ) give the location of the track to read.

( `dsk_cyl_t` ) "cyl\_expected" and "head\_expected" are used as the values search for in the sector headers.

Returns:

If successful, DSK\_ERR\_OK. Otherwise, a negative DSK\_ERR value.

( `dsk_cyl_t` ) only) If the driver does not support extended sector reads, then DSK\_ERR\_NOTIMPL will be returned.

#### 4.15 dsk\_lseek, dsk\_pseek

```

dsk_lseek(DISK_DEVICE dev, const DISK_GEOMETRY *geom, dsk_track_t track)
dsk_pseek(DISK_DEVICE dev, const DISK_GEOMETRY *geom, dsk_cyl_t cyl)

```

Seek to a given cylinder. Only the CPCEMU driver, the Linux floppy driver and the NTWDM floppy driver support this other drivers return DSK\_ERR\_NOTIMPL. You should not normally need to call these functions. They have been provided to support programs that emulate a uPD765A controller.

#### 4.16 dsk\_drive\_status

qsk\_eri\_t qsk\_qrlv6-ststus(D2K\_!DBI!EB 2eH' const D2K\_GEOMELBY \*g6om' qsk-bp6sq-t pes  
Get the drive's status (ready, read-only etc.). The byte returned will have one or more of the following bits set:

DSK\_ST3\_FAULT: Drive fault

DSK\_ST3\_RO: Read-only

DSK\_ST3\_READY: Ready

DSK\_ST3\_TRACK0: Head is over track 0

DSK\_ST3\_DSDRIVE: Drive is double-sided

DSK\_ST3\_HEAD1: Current head is head 1, not head 0. Usually this just depends on the value of the "head" parameter to this function.

Which bits will be "live" depends on which driver is in use, but the most trustworthy will be DSK\_ST3\_READY and DSK\_ST3\_RO. This function will never return DSK\_ERR\_NOTIMPL if the facility is not provided by the driver; a default version will be used.

#### 4.17 dsk\_dirty: Has drive been written to?

int qsk\_qrlv(D2K\_!DBI!EB 2eH)

This returns zero if the disc has not been written to since it was opened, nonzero if it has.

#### 4.18 dsk\_getgeom: Guess disc geometry

qsk\_eri\_t qsk\_g6t66om(D2K\_!DBI!EB 2eH' D2K\_GEOMELBY \*g6om)

This attempts to determine the geometry of a disc (number of cylinders, tracks, sectors etc.) by loading the boot sector. It understands DOS, Apple CP/M-86 and PCW boot sectors. If the geometry could be guessed, then "geom" will be initialised and DSK\_ERR\_OK will be returned. If no guess could be made, then DSK\_ERR\_BADFMT will be returned. Other values will result if the disc could not be read.

Some drivers (in particular the MYZ80 driver, and the Win32 driver under NT) only support certain fixed disc geometries. In this case, the geometry returned will reflect what the driver can use, rather than what the boot sector says

#### 4.19 dg\_\*geom : Initialise disc geometry from boot sector

qsk\_eri\_t qsk\_q666om(D2K\_GEOMELBY \*2eH' const nuz6neq cpar \*poot26ct)  
qsk\_eri\_t qsk\_b6m66om(D2K\_GEOMELBY \*2eH' const nuz6neq cpar \*poot26ct)  
qsk\_eri\_t qsk\_6bnuz66om(D2K\_GEOMELBY \*2eH' const nuz6neq cpar \*poot26ct)  
qsk\_eri\_t qsk\_6bnuz66om(D2K\_GEOMELBY \*2eH' const nuz6neq cpar \*poot26ct)

These functions are used by `dg_stdformat()`, but can also be called independently. Enter them with:

“self” is the structure to initialise

“bootsect” is the boot sector to initialise the structure.

Returns `DSK_ERR_BADFMT` if the sector does not contain a valid disc specification, or `DSK_ERR_OK` otherwise.

`dg_dosgeom` will check for a PC-DOS boot sector.

`dg_pcwgeom` will check for an Amstrad PCW boot sector.

`dg_cpm86geom` will check for a CP/M-86 boot sector.

`dg_aprigeom` will check for an Apricot DOS boot sector.

#### 4.20 `dg_stdformat` : Initialise disc geometry from a standard LibDsk format.

Initialises a `DSK_GEOMETRY` structure with one of the standard formats LibDsk knows about. Formats are:

**FMT\_180K:** 180k, 9 512 byte sectors, 40 tracks, 1 side

**FMT\_200K:** 200k, 10 512 byte sectors, 40 tracks, 1 side

**FMT\_CPCSYS:** Amstrad CPC system format - as `FMT_180K`, but physical sectors are numbered 65-73

**FMT\_CPCDATA:** Amstrad CPC data format - as `FMT_180K`, but physical sectors are numbered 193-201

**FMT\_720K:** 720k, 9 512 byte sectors, 80 tracks, 2 sides

**FMT\_800K:** 800k, 10 512 byte sectors, 80 tracks, 2 sides

**FMT\_1440K:** 1.4M, 18 512 byte sectors, 80 tracks, 2 sides

**FMT\_160K:** 160k, 8 512 byte sectors, 40 tracks, 1 side

**FMT\_320K:** As `FMT_160K`, but 2 sides

**FMT\_360K:** As `FMT_180K`, but 2 sides

**FMT\_720F:** As `FMT_720K`, but the physical/logical sector mapping is “round-back” rather than “alternate sides”. See section 3.1.1 for details.

**FMT\_1200F:** As `FMT_720F`, but with 15 sectors

**FMT\_1440F:** As `FMT_720F`, but with 18 sectors

**FMT\_ACORN160:** Acorn 40 track single sided 160k (used by ADFS 'S' format)

FMT\_ACORN320: Acorn 80 track single sided 320k (used by ADFS 'M' format)  
 FMT\_ACORN640: Acorn 80 track double sided 640k (used by ADFS 'L' format)  
 FMT\_ACORN800: Acorn 80 track double sided 800k (used by ADFS 'D' and 'E')  
 FMT\_ACORN1600: Acorn 80 track high density 1600k (used by ADFS 'F' format)  
 FMT\_BBC100 BBC micro 40 track single sided 100k (using FM encoding)  
 FMT\_BBC200 BBC micro 80 track single sided 200k (using FM encoding)  
 FMT\_MBEE400 Microbee 40 track double sided 400k  
 FMT\_MGT800 MGT 80 track double sided 800k (used by MGT D and Sam Coup ).

If the "fname" is not NULL, it will be pointed at a short name for the format (suitable for use as a program option see `tools/q2ktool.c` ).

If the "fdesc" is not NULL, it will be pointed at a description for the format. With these two, it's possible to enumerate geometries supported by the library without keeping a separate list in your program - see `tools/q2ktool.c` for example code that does this.

If additional formats have been specified in the libdskrc file (section 5.1), they will be returned by this function, using format numbers starting from the last builtin format plus 1.

#### 4.21 dsk\_\*\_forcehead: Override disc head

```
q2k_get_t q2k_set_forcehead(D2K_BDBIAEB zeh, int force)
q2k_get_t q2k_get_forcehead(D2K_BDBIAEB zeh, int *force)
```

(This function is deprecated it is equivalent to `dsk_setoption()` / `dsk_get_option()` with "HEAD" as the option name).

Forces the driver to ignore the head number passed to it and always use either side 0 or side 1 of the disc. This is used to read discs recorded on PC/CPC/Spectrum3 add-on 3.5 drives. Instead of the system software being programmed to use both sides of the disc, a switch on the drive was used to set which side was being used. Thus discs would end up with both sides saying they were head 0.

Anyway, when using `dsk_set_forcehead`, pass:

- 1: Normal - the head passed as a parameter to other calls is used.
- 0: Always use side 0.
- 1: Always use side 1.

#### 4.22 dsk\_\*\_option: Set/get driver option

```
q2k_get_t q2k_set_option(D2K_BDBIAEB zeh, const char *name, int value)
q2k_get_t q2k_get_option(D2K_BDBIAEB zeh, const char *name, int *value)
```

Sets or gets a driver-specific numeric option.

The "name" field is the option name. If the selected driver does not support the appropriate option, then the error DSK\_ERR\_BADOPT will be returned. If the option is valid but the value requested is not, DSK\_ERR\_BADVAL will be returned.

The following driver options are supported by the Linux and WDM floppy drivers:

**HEAD** Force the drive always to use one or other side of the disc, ignoring the disc geometry. Valid values are 0 or 1 to force one or other side of the disc, -1 to allow either.

**DOUBLESTEP** To support a 48tpi disc in a 96tpi drive, double all cylinder numbers. Valid values are 1 (enable) or 0 (disable).

**ST0 / ST1 / ST2 / ST3** These are the values of the floppy controller's 4 status registers returned by the last operation. They cannot be changed, only read.

The 'remote' driver supports the following option (plus any options that the remote driver supports):

**REMOTE:TESTING** This disables an optimisation in the remote driver, so that it sends method calls to the remote server even if it has been acknowledged. The purpose of this is to ensure that all calls to the remote driver result in RPC packets being sent.

#### 4.22.1 Filesystem driver options

It is possible that as part of its geometry probe, LibDsk will have detected a CP/M or DOS filesystem on a disc image. Alternatively, a disc image may contain filesystem metadata (for example, the YAZE ydisk and RCPMFS drivers both contain CP/M filesystem parameters). These parameters appear as driver options, prefixed with the name FS:. When making a copy, dsktrans enumerates the driver options on the source disc image and sets them to the same values on the destination image. This is necessary to ensure that (for example) when one YDSK is copied to another, its filesystem parameters are transferred. The current filesystem options supported by LibDsk are:

**FS:CP/M:BSH** Block shift - 3 = 1k, 4 = 2k, 5 = 4k...

**FS:CP/M:BLM** Block mask - (block size / 128) - 1

**FS:CP/M:EXM** Extent mask - roughly, how much does a directory entry cover?  
= 16k, 1 = 32k, 3 = 64k...)

**FS:CP/M:DSM** Number of data and directory blocks, minus 1

**FS:CP/M:DRM** Number of directory entries, minus 1

**FS:CP/M:AL0** Allocation bitmap of directory blocks (first 8 blocks)

**FS:CP/M:AL1** Allocation bitmap of directory blocks (second 8 blocks)

**FS:CP/M:CKS** Checksum vector size (normally (FS:CP/M:DRM - 1) / 4) can be 0x8000 for a fixed disc

**FS:CP/M:OFF** Number of boot tracks

**FS:CP/M:VERSION** Filesystem version (-2 (ISX), 2 (CP/M 2) or 3 (CP/M 3). This is only supported by the 'rcpmfs' driver.)

**FS:FAT:SECCLUS** Number of sectors per cluster

**FS:FAT:RESERVED** Number of reserved sectors

**FS:FAT:FATCOPIES** Number of FAT copies

**FS:FAT:DIRENTRIES** Number of root directory entries

**FS:FAT:MEDIABYTE** Media byte (usually the first byte of the FAT)

**FS:FAT:SECFAT** Number of sectors per FAT

Note that it is theoretically possible for a disc to have **FS:CP/M** and **FS:FAT** information - for example, a CP/M filesystem saved in a disc image ~~that~~ also contains FAT metadata, or vice versa.

#### 4.23 dsk\_option\_enum: Get list of driver options

```
qzk_gll_t qzk_option_enum(D2K_BDKIABK zell, int idx, char **options)
```

If "idx" is in the range 0 - number of driver options, (\*optname) is set to the name of the appropriate driver option. If not, (\*optname) is set to **NULL**.

#### 4.24 dsk\_\*\_comment: Set comment for disc image

```
qzk_gll_t qzk_set_comment(D2K_BDKIABK zell, const char *comment)
qzk_gll_t qzk_get_comment(D2K_BDKIABK zell, char **comment)
```

Used to get or set the comment (if any) for the current disc. ~~The~~ pointer passed or returned may be **NULL** (meaning "No comment"). The string ~~returned~~ belongs to LibDsk don't alter or free it.

#### 4.25 dsk\_type\_enum

```
qzk_gll_t qzk_type_enum(int idx, char **qlausame)
```

If "index" is in the range 0 - number of LibDsk drivers, (~~qlausame~~) is set to the short name for that driver (eg: "myz80" or "raw"). If not, (~~\*drvname~~) is set to **NULL**.

#### 4.26 dsk\_comp\_enum

```
qzk_gll_t qzk_comp_enum(int idx, char **combsame)
```

As ~~qzk\_type\_enum()~~, but lists supported compression schemes.

#### 4.27 dsk\_drvname, dsk\_drvdesc

```
const char *dsk_drvname(DISK_DRIVE dev)
const char *dsk_drvdesc(DISK_DRIVE dev)
```

Returns the driver name (eg: “myz80”) or description (eg “~~IBM~~ hard drive driver”) for an open disc image.

#### 4.28 dsk\_compname, dsk\_compdesc

```
const char *dsk_compname(DISK_DRIVE dev)
const char *dsk_compdesc(DISK_DRIVE dev)
```

Returns the compression system name (eg: “gz” NULL if the image isn’t compressed) or description (eg: “GZip compressed”) for an open image.

#### 4.29 dg\_ps2ls, dg\_ls2ps, dg\_pt2lt, dg\_lt2pt

Convert between logical sectors and physical cylinder/track/sector addresses. Normally these functions are called internally and you don’t need to use them.

```
uint32_t dg_ps2ls(const DISK_GEOMETRY *dev, uint32_t cyl, uint32_t head, uint32_t sector)
```

Converts physical C/H/S to logical sector.

```
uint32_t dg_ls2ps(const DISK_GEOMETRY *dev, uint32_t logical, uint32_t *cyl, uint32_t *head, uint32_t *sector)
```

Converts logical sector to physical C/H/S.

```
uint32_t dg_pt2lt(const DISK_GEOMETRY *dev, uint32_t cyl, uint32_t head, uint32_t track)
```

Converts physical C/H to logical track.

```
uint32_t dg_lt2pt(const DISK_GEOMETRY *dev, uint32_t logical, uint32_t *cyl, uint32_t *head)
```

Converts logical track to physical C/H.

#### 4.30 dsk\_strerror: Convert error code to string

```
char *dsk_strerror(int err)
```

Converts an error code returned by one of the other LibDisk functions into a printable string.

#### 4.31 dsk\_reportfunc\_set / dsk\_reportfunc\_get

```
void dsk_reportfunc_set(DSK_REPORTFUNC report, DSK_REPORTFUNC callback)
void dsk_reportfunc_get(DSK_REPORTFUNC *report, DSK_REPORTFUNC *callback)
```

Used to set callbacks from LibDsk to your own code, for LibDsk display messages during processing that may take time. The code could be used to set the text on the status line of your program window, for example.

```
callback void (*DSK_REPORTFUNC)(const char *message)
callback void (*DSK_REPORTFUNC)(void)
```

The first function you provide will be called when LibDsk wants to display a message (such as “Decompressing...”). The second will be called when the processing has finished.

#### 4.32 dsk\_set\_retry / dsk\_get\_retry

```
int dsk_set_retry(DSK_OPERATION op, int count)
int dsk_get_retry(DSK_OPERATION op, int *count)
```

Sets the number of times that a failed read, write, check or format operation will be attempted. 1 means “only try once, do not retry”.

#### 4.33 dsk\_get\_psh

```
int dsk_get_psh(int size, int *sectorsize)
```

Converts a sector size into the sector shift used by the uPIDE controller (eg: 128 - 0, 256 - 1, 512 - 2 etc.) You should not need to use this. The reverse operation is: sectorsize = (128 psh).

#### 4.34 dsk\_copy: Copy an entire disk image

```
int dsk_copy(DSK_OPERATION src, DSK_OPERATION dst, DSK_GEOMETRY *geom)
```

This will copy all data possible from one disk image to another. It only works on disk image files and so forth will return DSK\_ERR\_NOTIMPL.

The 'geom' parameter is usually left as NULL. Most disk images contain enough metadata that their structure can be determined unambiguously. Raw files (formats 'raw', 'rawoo', 'rawob' and 'logical') do not. If the source or the target is one of these files, the copy may fail with DSK\_ERR\_BADFMT. If so, 'geom' parameter should be passed, describing the layout to use.

### 4.35 Structure: DSK\_FORMAT

This structure is used to represent a sector header. It has members:

**fmt\_cylinder:** Cylinder number.

**fmt\_head:** Head number.

**fmt\_sector:** Sector number.

**fmt\_sectsize:** Sector size in bytes.

### 4.36 LibDsk errors

**DSK\_ERR\_OK:** No error.

**DSK\_ERR\_BADPTR:** A null or otherwise invalid pointer was passed to a LibDsk routine.

**DSK\_ERR\_DIVZERO:** Division by zero: For example, a DSK\_GEOMETRY is set to have zero sectors.

**DSK\_ERR\_BADPARM:** Bad parameter (eg: if a DSK\_GEOMETRY is set up with `geom_cylinders = 40`, trying to convert a sector in cylinder 65 to a logical sector will give this error).

**DSK\_ERR\_NODRVR:** Requested driver not found in `qsk_oben()` / `qsk_clet()`.

**DSK\_ERR\_NOTME:** Disc image could not be opened by requested driver.

**DSK\_ERR\_SYSERR:** System call failed. `errno` holds the reason.

**DSK\_ERR\_NOMEM:** `msjoc()` failed to allocate memory.

**DSK\_ERR\_NOTIMPL:** Function is not implemented (eg, this driver doesn't support `qsk_xlseq()`).

**DSK\_ERR\_MISMATCH:** In `qsk_lcpck()` / `qsk_bcpck()`, sectors didn't match.

**DSK\_ERR\_NOTRDY:** Drive is not ready.

**DSK\_ERR\_RDONLY:** Disc is read-only.

**DSK\_ERR\_SEEKFAIL:** Seek fail.

**DSK\_ERR\_DATAERR:** Data error.

**DSK\_ERR\_NODATA:** Sector ID found, but not sector data.

**DSK\_ERR\_NOADDR:** Sector not found at all.

**DSK\_ERR\_BADFMT:** Not a valid format.

**DSK\_ERR\_CHANGED:** Disc has been changed unexpectedly.

**DSK\_ERR\_ECHECK:** Equipment check.

**DSK\_ERR\_OVERRUN:** Overrun.

**DSK\_ERR\_ACCESS:** Access denied.

**DSK\_ERR\_CTRLR:** Controller failed.

**DSK\_ERR\_COMPRESS:** Compressed file is corrupt.

**DSK\_ERR\_RPC:** Error in remote procedure call.

**DSK\_ERR\_BADOPT:** Driver does not support the requested option.

**DSK\_ERR\_BADVAL:** Driver does support the requested option, but the passed real is out of range.

**DSK\_ERR\_UNKNOWN:** Unknown error

### 4.37 Miscellaneous

**LIBDSK\_VERSION** is a macro, defined as a string containing library version - eg "1.0.0"

## 5 Initialisation files

In addition to its built-in library of formats, LibDsk can load formats from one or two external files - a systemwide file (libdiskrc) and a user-specific file (.libdiskrc). The rules for how these files are found differ from platform to platform.

### 5.1 libdiskrc format

The file format is similar to a Windows .INI file. Each format is described in a section, which starts with the format name in square brackets (format names may not start with a hyphen). After the format name, there are a number of lines of the form variable=value.

Anything after a semicolon or hash character is treated as a comment and ignored. Blank lines are also ignored.

For each geometry, the entries listed below can be present, not all the values are present, LibDsk will use default values from its pcw180 format. As you can see, they correspond to members of the DSK\_GEOMETRY structure.

**description=DESC** The description of the format as shown by (for example) `dskinfo help`.

**sides=TREATMENT** How a double-sided disk is handled. This can either be `alt` (sides alternate - used by most PC-hosted operating systems), `outback` (use side 0 tracks 0-79, then side 1 tracks 79-0 - used by 144FEAT CP/M disks), `outout` (use side 0 tracks 0-79, then side 1 tracks 0-79 - used by some `corn` formats) or `extsurface` (sectors on side 0 are numbered 1, sectors on side 1 are numbered n1 - n\*2). If the disk is single-sided, this parameter can be omitted.

**cylinders=COUNT** Sets the number of cylinders (usually 40 or 80).

**heads=COUNT** Sets the number of heads (usually 1 or 2 for single- or double-sided).

**sectors=COUNT** Sets the number of sectors per track.

**secbase=NUMBER** Sets the first sector number on a track. Usually 1 some Acorn formats use 0.

**secsize=COUNT** Sets the size of a sector in bytes. This should be a power of 2.

**datarate=VALUE** Sets the rate at which the disk should be accessed. This is one of HD, DD, SD or ED.

**rwgap=VALUE** Sets the read/write gap.

**fmtgap=VALUE** Sets the format gap.

**recmode=FM or MFM** Sets the recording mode - FM or MFM. For backward compatibility, the alternate syntax **FM=Y or N** is also supported.

**complement=Y or N** Sets the complement flag - Y if the format stores data complemented.

**multitrack=Y or N** Sets multitrack mode.

**skipdeleted=Y or N** Sets whether to skip deleted data.

### 5.1.1 libdiskrc example

```
This is FMT_800K as a libdiskrc entry
xc15q
Description = 800K XCE3DD format
Sides = 4
Cylinders = 80
Heads = 3
Sectors = 10
SectorSize = 1
DataRate = 2D
RMCSB = 13
FmtCSB = 33
RecMode = MFM
xc15
Description = 500K XCE3 format
Cylinders = 40
```

... etc.

The supplied libdiskrc.sample file contains libdiskrc-format definitions of all the built-in disk formats.

## 5.2 Locating libdiskrc

### 5.2.1 UNIX

The systemwide file is located at `datadir/LibDsk/libdiskrc`. The `datadir` is usually `/usr/local/share` you can change it with the `datadir=prefix` arguments to the `configure` script.

The user-specific file is `(HOME)/.libdiskrc`.

### 5.2.2 Win32

The systemwide file is in the path specified at

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Shell\libdskrc
```

If this registry key is not found, LibDsk finds the path of the program that called it (using `GetModuleFileName()`), and then uses “`../../programdata/share/libdskrc`”.

The user-specific file is in the path specified at

```
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Shell\libdskrc
```

If this registry key is not present, the user’s “My Documents” directory is used. Either way, the file is called `.libdskrc`.

### 5.2.3 Win16

The systemwide file is found from the location of the calling program using `GetModuleFileName()`. There is no user-specific file.

### 5.2.4 DOS

The systemwide file is only searched for if the `LIBDSK` environment variable is set if it is set, it is assumed to be the name of the directory containing `libdskrc`. There is no user-specific file.

## 6 Reverse CP/M-FS (rcpmfs) backend

The `rcpmfs` backend is designed to present a host directory as read/write CP/M disk image. This has a number of uses:

You could construct a CP/M disk image using `disktrans` directory filename

Conversely, you could extract the files from a CP/M disk image using `disktrans` filename directory

It is possible for a CP/M emulator running a genuine copy of CP/M to use LibDsk to access files on the host system, without altering the BDOS by installing additional drivers.

`rcpmfs` does not work with systems that only support “8.3” filenames it also needs a system call that can set the size of a file (such as `truncate()` under UNIX). It therefore remains unimplemented in the DOS and Win16 versions of the library.

### 6.1 In Use

To use an `rcpmfs` directory in LibDsk, pass a directory name instead of a filename. Files in the directory which match CP/M naming conventions (8.3 filenames) will appear in the emulated disk image if there are more files than will fit the emulated disk, LibDsk will stop when it reaches one that doesn’t fit. Under Windows, the ‘short filename’ is used, so files with names not matching CP/M conventions may also be mapped with names like `README1.HTM`.

CP/M has 16 user areas (some variants support 32 rcpmfs ~~domains~~), and files with the same name can exist in each area. rcpmfs represents ~~non-user~~ user areas by prepending “nn..” to the filename so if a CP/M program created a file ~~called~~ EXAMPLE.DAT in user 4, this would be saved as “04..example.dat” in the ~~current~~ directory. The double dot ensures that the resulting filename is not a valid ~~CP/M~~ name, and therefore won’t conflict with any file in user 0.

rcpmfs can behave as a CP/M 2 or CP/M 3 filesystem. If the ~~latter~~ constructs a disc label (based on the name of the directory) and turns ~~on~~ time stamping. Update and access stamps are used, because they map nicely to the ~~time~~ system call. It can also emulate the filesystem used by the ISX emulator, which ~~uses~~ file sizes slightly differently.

## 6.2 rcpmfs initialisation file

For a directory to be usable by rcpmfs, it should contain a ~~filled~~ .libdisk.ini describing the format to use. This file is in INI format, similar to ~~diskrc~~ (section 5.1). It must contain only one section: RCPMFS. Within that section, the following variables may be present:

**BlockSize** Size of a CP/M data block. Must be a power of 2, and at least 1024. If there are more than 255 blocks in the CP/M filesystem, this must be at least 2048.

**DirBlocks** Number of blocks containing the CP/M directory.

**TotalBlocks** Total number of data and directory blocks.

**SysTracks** Number of system tracks. These will be stored in a file called ~~libdisk~~.boot.

**Version** CP/M version that will be accessing the filesystem. This should be 2, 3 or ISX:

2 CP/M 2 no time stamps or disk labels.

3 CP/M 3 time stamps and disk labels are present.

ISX Used by the ISX emulator. Similar to CP/M 2, but byte 13 of the ~~CP/M~~ directory entry holds the number of ~~used~~ bytes in the last record, not the number of used bytes.

**Format** Name of one of the LibDsk built-in or user-supplied formats, giving the geometry that the simulated disk will have. Alternatively, you can specify the format manually, using the same variable names as in libdiskrc.

If there is no .libdisk.ini file present, LibDsk will assume ~~BlockSize=1024, DirBlocks=2, TotalBlocks=175, SysTracks=1, Version=3, Format=pcw180~~

If you call dsk\_option\_set with any of the following options

FS:CP/M:BSH

FS:CP/M:BLM

FS:CP/M:DSM

FS:CP/M:DRM

FS:CP/M:OFF

FS:CP/M:VERSION

and the value written differs from the one used before, then a new .libdsk.ini file will be written with the revised filesystem parameters and the disk rescanned. This allows a command of the form:

```
format -o:lyb6 rcpmfs qisc-image qirecfol7
```

to stand a reasonable chance of working as long as the source image has a CP/M filesystem that LibDsk can detect.

To select ISX format using dsk\_option\_set(), use -2 as the filesystem version:

```
format -o:lyb6 rcpmfs qisc-image qirecfol7 -2
```

### 6.3 Bugs

repmfs is not without its bugs and missing features:

To my knowledge, repmfs has only been tested under the `disk` pattern of usage (which writes the directory and then the file space) with fairly simple operations in a CP/M emulator. It is not known how well it holds up under heavy use as a live CP/M filesystem.

The CP/M attributes F1-F4, passwords and permissions are not mapped. The SYS and ARC attributes are only mapped in the Win32 version.

Formatting (or reformatting) an repmfs directory writes a new .libdsk.ini containing the geometry used to do the format. However, since `DISKGEOMETRY` doesn't contain the CP/M filesystem parameters (block size, block count, etc.) these will be the ones previously used in that directory, and quite possibly completely wrong. If you want to 'format' the directory using LibDsk, call `dsk_set_option()` with the six "FS:CP/M:" options listed above to set up the correct filesystem parameters. Or create the .libdsk.ini by other means.

## 7 LibDsk under Windows

This section mainly deals with the subject of direct floppy access. Other aspects of LibDsk remain relatively consistent across Windows versions.

As with so many other aspects of Windows, direct access to floppy drive is a case of "write once - debug everywhere". Not only does support vary across different systems, it varies depending on whether LibDsk was compiled with a 16-bit compiler or a 32-bit one. This table shows the different possibilities and the resulting behaviour:

Windows Version	Win16 Subsystem	Win32 Subsystem
3.x	Fairly good	n/a
4.x (95, 98 and ME)	Good but less stable	Limited
NT, 2000, XP	Very limited	
2000, XP ntwdm driver	Good	

<sup>1</sup>Originally said by Microsoft with respect to ava. Pot. ~~Kit~~ Black.

## 7.1 Windows 3.x

Only the 16-bit build of LibDsk will run. The floppy support in Win16 is pretty much the same as in DOS there is support for discs with arbitrary numbers of tracks and sectors, and arbitrary sector sizes. This means that LibDsk, for example, read Acorn ADFS floppies.

## 7.2 Windows 4.x (95, 98 and ME)

Both the 16-bit and 32-bit versions of LibDsk will run. The 16-bit version is more capable, but less stable it can read Acorn ADFS floppies, while the 32-bit version cannot. Unfortunately, 32-bit programs can't link to the 16-bit version of LibDsk, but there is a workaround (described below) involving the use of LDRSERVER.

## 7.3 Windows NT (NT 3.x, NT 4.x, 2000, XP) without ntwdm driver

The floppy drive can only read/write formats which are supported by the floppy driver. This is the case using either version of LibDsk.

## 7.4 Windows 2000 and XP with ntwdm driver

Simon Owen's enhancement to the Windows 2000 floppy driver can be downloaded from <http://simonowen.com/fdrawcmd/>. Once it is installed, LibDsk (using its 'ntwdm' driver rather than 'floppy') has pretty much carte blanche handling floppy formats, and can access discs in many formats including Acorn ADFS.

## 7.5 General comments on programming floppy access for Windows

LibDsk has four independent drivers for accessing floppy disks under Windows. They are:

### 7.5.1 The Win16 driver.

This uses INT 0x13 to do the reads and writes, just as in MSDOS. Again as in MSDOS, there is a diskette parameter table pointed to by INT 0x1E. This table seems not to be documented, which is perhaps why the Win16 subsystem in Windows 2000/XP doesn't implement it. You can, fortunately, tell if this is the case if the first two bytes are both 0xC4, then what you have is a Windows 2000 trap rather than a diskette parameter table.

### 7.5.2 The Win32c driver.

This driver uses VWIN32 services to make INT 0x13-style calls under Windows 9x. However, there is no VWIN32 call to change the diskette parameter table, which is why the Win16 driver can do things the Win32 drivers can't. It's not possible to get round this by thunking to a 16-bit DLL either the INT 0x1E vector is zero for 16-bit DLLs in 32-bit processes.

---

<sup>2</sup>And no, the Generic Thunk isn't good enough. I've tried it.

### 7.5.3 The Win32 driver.

Windows NT gets close (but not close enough) to the UNIX idea that everything is a file. So while in theory it would be enough to use the normal Win32 driver on “v:” “v:”, in practice there are a number of nasty subtleties relating to such things as memory alignment and file locking.

### 7.5.4 The ntwdm driver.

This driver is a wrapper around fdrawcmd.sys, which allows commands to be issued to the floppy controller.

### 7.5.5 Other floppy APIs

Sydex produce a replacement floppy driver for 32-bit versions of Windows (SydexFDD) which is not supported by LibDsk.

## 7.6 LDSERVER

LDSERVER is a program that makes the 16-bit LibDsk DLL available to 32-bit programs. It does this by creating a mailslot (“\msg100\LIBDsk”) and listening for messages. Each message corresponds to a LibDsk call.

The 32-bit LibDsk library checks for this mailslot and, if it finds it, uses it in preference to its own floppy support.

### 7.6.1 Compiling LDSERVER

A compiled version of LDSERVER is not supplied. You will need to build it yourself from the files in the rpcserv directory projects are provided for Microsoft Visual C 1.5 and Borland C 5.0.

LDSERVER calls functions in NETAPI.DLL. If your compiler doesn't include an import library for this DLL, you will have to generate it using the IMPLIB tool - eg:

```
IMPLIB IELVYITV IELVY'DGT
```

or the equivalent utility for your compiler.

### 7.6.2 Using LDSERVER

Just run LDSERVER.EXE, and then use a 32-bit LibDsk program. The server window shows a reference count (0 if it is idle, nonzero if LibDsk programs are using it) and the status should change to “Active” when it is performing disk access.

LDSERVER does not shut down automatically.

### 7.6.3 Important Security Warning

LDSERVER is a 16-bit program, written using APIs intended to use on a local area network. These APIs have no security support. It will happily obey commands sent from anywhere on your network. If your computer is connected to the Internet, it will obey commands sent to it over the Internet. A malicious hacker could use LDSERVER to overwrite important system files or read confidential documents.

If you have a firewall, then make sure that the NetBIOS ports 138 and 139 are blocked. If you don't have a firewall do not run LDSERVER while your computer is connected to the Internet

## 7.7 LibDsk and COM

If you are building the 32-bit version of LibDsk with Visual C++, you can also build the accompanying 'atlibdsk' project, which builds a version of LibDsk that exports its API through COM. This allows relatively easy use of LibDsk for languages that support COM binding, such as Visual BASIC or .NET languages.

### 7.7.1 General points

Where LibDsk functions return a `dsk_err_t`, ATLIBDSK returns a COM HRESULT. This will be `S_OK` for success, a general COM error (such as `E_POINTER` or `E_INVALIDARG`), or a `FACILITY_ITF` error (`0x8004xxxx`). The low word of a `FACILITY_ITF` error is the LibDsk error code, converted to a positive number (eg: `0x8004000C` is `FACILITY_ITF` error 12, so the LibDsk error is -12, `DSK_ERR_SEEKERR`).

Sector buffers to be read/written must be passed as variants containing arrays of bytes.

The arrays of `DSK_FORMAT` structures passed to `dsk_lformat` (`dsk_pformat`) are replaced by variants containing arrays of bytes - four bytes per sector to format. The last byte is the physical sector shift (0 for 128, 1 for 256).

ATLIBDSK exports four object classes:

### 7.7.2 Library

This contains LibDsk functions not associated with a particular disk image. Its methods are:

Method	Equivalent LibDsk call	Comments
<code>open</code>	<code>dsk_open</code>	Instantiates a new Disk object.
<code>create</code>	<code>dsk_creat</code>	Instantiates a new Disk object.
<code>get_psh</code>	<code>dsk_get_psh</code>	
<code>dosgeom</code>	<code>dg_dosgeom</code>	Instantiates a new Geometry object.
<code>cpm86geom</code>	<code>dg_cpm86geom</code>	Instantiates a new Geometry object.
<code>pcwgeom</code>	<code>dg_pcwgeom</code>	Instantiates a new Geometry object.
<code>aprigeom</code>	<code>dg_aprigeom</code>	Instantiates a new Geometry object.
<code>stdformat</code>	<code>dg_stdformat</code>	Instantiates a new Geometry object.
<code>stdformat_count</code>		Returns the number of formats supported by <code>stdformat</code> .
<code>type_enum</code>	<code>dsk_type_enum</code>	Returns TRUE if the passed index is valid, else FALSE.
<code>comp_enum</code>	<code>dsk_comp_enum</code>	Returns TRUE if the passed index is valid, else FALSE.
<code>reporter</code>	<code>dsk_reportfunc_set,get</code>	This is a property of type <code>IReporter</code> .

### 7.7.3 Geometry

This corresponds to the `DSK_GEOMETRY` structure. The following properties correspond to the structure members:

`sidedness`

`cylinders`

**heads**  
**sectors**  
**secbase**  
**datarate**  
**secsize**  
**rwgap**  
**fmtgap**  
**fm**  
**nomulti**  
**noskip**

There are also five functions. Four are for logical/physical sector conversions:

**ls2ps**  
**lt2pt**  
**ps2ls**  
**pt2lt**

and the last is `stdformat()`, which wraps `dg_stdformat()`.

#### 7.7.4 Disk

The Disk object corresponds to a LibDsk `DSK_PDRIVER` value. ~~You~~ should not create one yourself (method calls will fail with `E_POINTER`) but use the 'create' or 'open' methods of the Library object.

Functions included are:

**get\_geometry**  
**close**  
**drive\_status**  
**pread**  
**lread**  
**xread**  
**pwrite**  
**lwrite**  
**xwrite**  
**pcheck**

**lcheck**  
**xcheck**  
**pformat**  
**lformat**  
**apform**  
**alform**  
**ptread**  
**ltread**  
**xtread**  
**psecid**  
**lsecid**  
**lseek**  
**pseek**  
**option\_enum**

all of which are pretty similar to their LibDsk namesakes. There are also the following properties:

**comment**  
**option**  
**retries**  
**drvname**  
**drvdesc**  
**compname**  
**compdesc**

#### 7.7.5 IReporter

IReporter is used for the LibDsk message callback. It is an interface that should be implemented by an object in your program. Set the library's reporter property to your object then its report() and endreport() methods will be called.

## 8 LibDsk RPC system

The LibDsk RPC system is designed to make disc drives on remote computers transparently available to LibDsk applications. It operates on a client/server basis. LibDsk contains a driver (called 'remote') that can act as a client and it can be used to implement a server.

The on-the-wire protocol is described in protocol.txt in the documentation directory.

## 8.1 The 'serial' driver

This is designed for using LibDsk over a serial connection from a 3.5" computer to a 5.25" computer. The filename specification to use at the remote end is:

```
remotename remotetype remotecompress
```

for example:

```
remotename:\dev\ttyS0:000crtsets:7:
```

The various parts of this filename specification are:

**port** The local serial port to use.

Under Linux, this is the name of a serial port (eg /dev/ttyS0)

Under Windows, this is likewise the name of a serial port (COM1:).

Under DOS, you need to have a FOSSIL serial port driver loaded. LibDsk was tested using ADF [http://ftp.iis.com.br/pub/simtel1/msdos/fossil/adf\\_150.zip](http://ftp.iis.com.br/pub/simtel1/msdos/fossil/adf_150.zip) (or do a web search for adf\_150.zip). The port is then the number assigned by the FOSSIL driver (normally 0). Note also that ADF uses a fixed baud rate, so you should make sure that the rate on the remote line matches the rate that was used when ADF was loaded.

**baud** The speed and handshaking options. LibDsk does not allow the number of bits, the parity or the count of stop bits to be changed it insists on 8-bit communications with 1 stop bit and no parity. The speed is a number (300, 1200 etc.) and the handshake option is "crtsets" (to use RTS/CTS handshaking) or "-crtsets" (not to). If neither handshake option is present "crtsets" is assumed.

**remotename** The name of the file or drive on the remote computer.

**remotetype** The type of the file/drive ("disk", "floppy" etc.).

**remotecompress** The compression to use on the remote computer.

### 8.1.1 Servers for the serial driver

One of the sample utilities supplied with LibDsk is called **server** (serslave.exe under DOS / Windows). This is a server using the same serial protocol as above.

Launch serslave with the command:

```
server remotename remotetype
```

for example:

```
server COM1:000crtsets
```

or in DOS (again, a FOSSIL driver is required):

```
server 0:12500
```

I have written a similar server for CP/M systems, called **AUXD**. This is a separate download from the LibDsk web page.

## 8.2 The 'fork' driver

The 'fork' driver is used (on any system which supports the `fork()` system call) to send LibDsk requests to a local program using pipes. This driver is written for testing purposes, but may come in handy as a poor man's plugin system. The filename specification is:

```
format: remotename remotetype remotecompress
```

for example:

```
format:\device\program\type\compress
```

The various parts of this filename specification are:

**program** The name of the program to use `execp()` is used to launch it. If no path is given the user's PATH will be searched. The program must call LibDsk calls from its standard input and send results to its standard output.

**remotename** The name of the file or drive.

**remotetype** The type of the file/drive ("disk", "floppy" etc.).

**remotecompress** The compression to use.

An example of a server for this protocol is the example 'fork' program this is a very simple wrapper around `dsk_rpc_server()` which reads packets from its standard input and writes them to its standard output.

## 9 Writing new drivers

The interface between LibDsk and its drivers is defined by the `DRV_CLASS` structure. To add a new driver, you create a new `DRV_CLASS` structure and add it to various files.

There are two methods of writing a driver. One is to provide the functions yourself. The other is to write a driver using LDBS as its internal storage format in that case, you need only provide `dc_open()`, `dc_create()` and `dc_close()`. The latter technique is particularly suited to formats that can't be written in place (so you have to parse the whole file on open and rewrite the whole file on close anyway), and is described in section 10.

Assuming you want to create a full driver that rewrites itself proceed as follows:

### 9.1 The driver header

Firstly, create a header for this driver, basing it on (for example) `libdsk\drv\header.h`. The first thing in the header (after the LGPL banner) is:

```
typedef struct
{
    DSK_DRIVER *driver;
    FILE *file;
    int fd;
    int *fd;
} DRV_HEADER;
#define DSK_DRIVER DRV_HEADER
```

This is where you define any variables that your driver needs to store for each disc image. In the case of the “raw” driver, this consists of a FILE pointer to access the underlying disc file, a “readonly” flag, and the current size of the drive image file. The first member of this structure must be of type DSK\_DRIVER.

The rest of this header consists of function prototypes, which will come back to later.

## 9.2 The driver source file

Secondly, create a .c file for your driver. Again, it’s probably easiest to base this on lib/drvposix.c. At the start of this file, create a DRV\_CLASS structure, such as:

```

DBA_CIV22 qc_posix =
    {
        sizeof(DSK_DRIVER),
        NULL,
        "lib/drvposix",
        "posix-open",
        "posix-close",
        "posix-close"
    }

```

The first four entries in this structure are:

The size of your driver’s instance data

The driver’s superclass. This should be left as NULL.

Possible names for the driver (each will be matched against the name passed to qc-open() / qc-close()). Each possible name, including the last, must be followed by 0.

The driver’s description string.

The remainder of the structure is composed of function pointers the types of these are given in drv.h. At the very least, you will need to provide the first three pointers (\*\_open, \*\_creat and \*\_close) to make the driver vaguely useful, you will also need to implement some of the others.

Once you have created this structure, edit:

drivers.h. Add a declaration for your DRV\_CLASS structure such as

```
extern DBA_CIV22 qc_posix;
```

drivers.inc. Insert a reference to your structure (eg: “qc\_posix”) in the list. Note that order is important the comments in drivers.inc describe how to decide where things go.

Edit “lib/Makefile.am”. Near the top of this file is a list of drivers and their header files just add your .c and .h to this list.

If your driver depends on certain system headers (as all the other drivers do) then you will need to add checks for these to “configure.in” and “lib/rvi.h” then run “autoconf” to rebuild the configure script.

The function pointers in the DRV\_CLASS structure are described in drv.h. The first parameter to all of them ("self") is declared as a pointer to DRV\_DRIVER. In fact, it is a pointer to the first member of your instance data structure. You must cast the pointer to the correct type:

```
\* Example: Is this meant for our driver? *\
if (self->class == DC_DRIVER) return DSK_ERR_BADFILE
bx26f = (FOO_DSK_DRIVER *)self
```

and you're in business.

### 9.3 Driver functions

#### 9.3.1 dc\_open

```
int (*dc_open)(DRV_DRIVER self, const char *filename)
```

Attempt to open a disc image. Entered with:

"self" points to the instance data for this disc image (see above) it will have been initialised to zeroes using memset().

"filename" is the name of the image to open.

Return:

DSK\_ERR\_OK: The driver has successfully opened the image.

DSK\_ERR\_NOTME: The driver cannot handle this image. Other drivers should be allowed to try to use it.

other: The driver cannot handle this image. No other drivers should be tried (eg: the image was recognised by this driver, but is corrupt).

If the file has a comment, record it here using dsk\_set\_comment()

#### 9.3.2 dc\_creat

```
int (*dc_creat)(DRV_DRIVER self, const char *filename)
```

Attempt to create a new disc image. For the "floppy" drivers it behaves exactly as dc\_open. Parameters and results are the same as for dc\_open except that DSK\_ERR\_NOTME is treated like any other error.

#### 9.3.3 dc\_close

```
int (*dc_close)(DRV_DRIVER self)
```

Close the disc image. This will be the last call your driver will receive for a given disc image file, and it should free any resources it is using. Whether it returns DSK\_ERR\_OK or an error, this disc image will not be used again.

### 9.3.4 dc\_read

qsk\_glt\_t (\*dc\_read)(DSK\_DRIVE dev, const DSK\_GEOMETRY \*geom, void \*buf, qsk\_bcl\_t

Read a sector. Note that sector addresses passed to drivers are always in C/H/S format. This function has the same parameters and return values as dsk\_read().

You don't need to check the RECMODE\_COMPLEMENT flag in the geometry structure (this applies to all read and write functions). If the flag is set, the LibDsk core will complement the results from the driver before returning them to the caller. Similarly, any buffer passed for a write will already be complemented if appropriate.

### 9.3.5 dc\_write

qsk\_glt\_t (\*dc\_write)(DSK\_DRIVE dev, const DSK\_GEOMETRY \*geom, const void \*buf, qsk\_bcl\_t

Write a sector. This function has the same parameters and return values as dsk\_pwrite(). If your driver is read-only, leave this function pointer NULL.

### 9.3.6 dc\_format

qsk\_glt\_t (\*dc\_format)(DSK\_DRIVE dev, const DSK\_GEOMETRY \*geom, qsk\_bcl\_t cyl, head,

Format a track. This function has the same parameters and return values as dsk\_pformat(). If your driver cannot format tracks, leave this function pointer NULL.

### 9.3.7 dc\_getgeom

qsk\_glt\_t (\*dc\_getgeom)(DSK\_DRIVE dev, DSK\_GEOMETRY \*geom)

Get the disc geometry. Leave this function pointer as NULL or either:

1. Your disc image does not allow a caller to use an arbitrary geometry. The drivers which currently do this are the Win32 one, because Windows NT decides on the geometry itself and doesn't let programs change it, and the MYZ80 and SIMH ones, which have a single fixed geometry.
2. Your disc image file contains enough information to populate a DSK\_GEOMETRY completely. The rcpmfs and ydsk drivers do this.
3. You want to do an extended geometry probe including a call to the default one. The internal function dsk\_defgetgeom() has been provided for this it's the same as dsk\_getgeom() but always uses the standard probe. The LibDsk driver does this.

Returns DSK\_ERR\_OK if successful DSK\_ERR\_NOTIME or DSK\_ERR\_NOTIMPL to fall back to the standard LibDsk geometry probe otherwise to indicate failure.

### 9.3.8 dc\_secid

qsk\_glt\_t (\*dc\_secid)(DSK\_HEADER \*h, const DSK\_GEOMETRY \*geom, qsk\_bcyl\_t cylinder, qsk\_head\_t head, qsk\_secid\_t \*secid)

Read the ID of the next sector on a certain track/head, and return the "result". Ideally you would simulate a rotating disc, so that the IDs are returned in the same order that they were written when the disc was formatted. This function is used to test for discs in CPC format (which have oddly-numbered physical sectors) if the disc image can't support this (eg: the "raw" or Win32 drivers) then leave the function pointer NULL.

### 9.3.9 dc\_xseek

qsk\_glt\_t (\*dc\_xseek)(DSK\_HEADER \*h, const DSK\_GEOMETRY \*geom, qsk\_bcyl\_t cylinder, qsk\_head\_t head, qsk\_secid\_t \*secid)

Seek to a given cylinder / head. For disc images, just return `DSK_ERR_OK` if the cylinder/head are in range, or `DSK_ERR_SEEKFAIL` otherwise. For a floppy driver, only implement this function if your FDC can perform a seek itself.

### 9.3.10 dc\_xread, dc\_xwrite

qsk\_glt\_t (\*dc\_xread)(DSK\_HEADER \*h, const DSK\_GEOMETRY \*geom, void \*buf, qsk\_bcyl\_t cylinder, qsk\_head\_t head, qsk\_secid\_t \*secid, int \*deleted)  
qsk\_glt\_t (\*dc\_xwrite)(DSK\_HEADER \*h, const DSK\_GEOMETRY \*geom, const void \*buf, qsk\_bcyl\_t cylinder, qsk\_head\_t head, qsk\_secid\_t \*secid, int \*deleted)

Read / write sector whose ID may not match its position on disc which is marked as deleted. Only implement this if your disc image emulator or your floppy driver exposes this level of functionality. Currently it is implemented in the Linux, NTWDM and CPCEMU drivers, plus those using LDBS as their internal storage (LDBS itself, DSK, EDSK, ApriDisk, CFI, V3, CopyQM, QRST, Teledisk).

### 9.3.11 dc\_status

qsk\_glt\_t (\*dc\_status)(DSK\_HEADER \*h, const DSK\_GEOMETRY \*geom, qsk\_bcyl\_t cylinder, qsk\_head\_t head, int \*status)

Return the drive status (see `dsk_drive_status()` for the bit return). "result" will contain the value calculated by the default implementation on most image file drivers, all you have to do is set the read-only bit if appropriate.

### 9.3.12 dc\_tread

qsk\_glt\_t (\*dc\_tread)(DSK\_HEADER \*h, const DSK\_GEOMETRY \*geom, void \*buf, qsk\_bcyl\_t cylinder, qsk\_head\_t head, qsk\_secid\_t \*secid)

Read a track. You need only implement this if your floppy driver exposes the relevant functionality if you don't, the library will use multiple calls to `dc_read()` instead. This function has the same parameters and return values as `dsk_read()`.

### 9.3.13 dc\_xtread

qzk\_6ll\_t (\*qc\_xl69q)(D2K\_DBIAEB \*z6ll' const D2K\_GEOMELKX \*g6om' void \*pnt' qzk\_bclj\_t

Read a track, with extended sector matching (sector headers on disc differ from physical location). This function has the same parameters and return values as dsk\_xtread(). As with dc\_tread(), you need only implement this function if your floppy driver has a special READ TRACK command.

### 9.3.14 dc\_option\_enum

qzk\_6ll\_t (\*qc\_obqom\_enuu)(D2K\_DBIAEB \*z6ll' int iqz' char \*\*obqum6)

List numerical options which your driver supports. If your driver does not support any, you need not implement this.

### 9.3.15 dc\_option\_set, dc\_option\_get

qzk\_6ll\_t (\*qc\_obqom\_z6t)(D2K\_DBIAEB \*z6ll' const char \*obqum6' int v6lue)  
qzk\_6ll\_t (\*qc\_obqom\_g6t)(D2K\_DBIAEB \*z6ll' const char \*obqum6' int \*v6lue)

Get or set the value of a numerical option. Again, if your driver has no numerical options, this need not be implemented.

Note that numerical options can 'belong' either to a driver or to the LibDsk core, with the driver taking priority. For example:

If LibDsk accesses a FAT-format disc image using the 'dsk' driver, neither LibDsk nor the driver will support the FS:CP/M:BSH option.

If LibDsk accesses a CP/M-format disc image using the 'dsk' driver, dsk\_get\_geometry() will detect the CP/M filesystem. Since the driver does not support the FS:CP/M:BSH option, it will be handled by the LibDsk core.

If LibDsk accesses a CP/M-format disc image using the 'yds' driver, the driver does support the FS:CP/M:BSH option and so it will be handled by the driver.

It is possible for a driver to rely on the option support in LibDsk core rather than implement its own. This means a lot less code needs to be written but it does not allow any validation to be performed on the values an option can hold or does it notify the driver when the value of an option is changed. Currently this system is used by the myz80 driver.

To use this system, create the variables you require with `disk_option`:

qzk\_6ll\_t qzk\_iz6toqom(D2K\_DBIAEB \*z6ll' const char \*obqum6' int v6lue' int cl69t6)

The first three parameters are the same as for `dsk_set_option`. The last should be set to 1 to create the new variable, or 0 to return `DSK_ERR_BADOPT`. The variable is not present.

To read a value back, use `dsk_get_option()` as normal.

### 9.3.16 dc\_trackids

qzk\_ert\_t (\*qc\_tkqid)(D2K\_DBIAEB \*zejt, const D2K\_GEOMELBA \*geom, qzk\_bclj\_t clyudq)

Read the IDs of all sectors on the specified track, preferably in the correct order and starting at the index hole. If you leave this function pointer as NULL, LibDsk will use a default implementation.

### 9.3.17 dc\_rtread

qzk\_ert\_t (\*qc\_tkqid)(D2K\_DBIAEB \*zejt, const D2K\_GEOMELBA \*geom, void \*pnt, qzk\_bclj\_t)

For future expansion. Leave this function pointer as NULL.

### 9.3.18 dc\_to\_ldbs

qzk\_ert\_t (\*qc\_tkqid)(D2K\_DBIAEB \*zejt, struct lqps \*lqps, D2K\_GEOMELBA \*geom)

Export the current disk image file as an LDBS blockstore. If driver is not for a disk image file, there's no need to implement this function.

### 9.3.19 dc\_from\_ldbs

qzk\_ert\_t (\*qc\_tkqid)(D2K\_DBIAEB \*zejt, struct lqps \*lqps, D2K\_GEOMELBA \*geom)

Replace the entire contents of this disk image with the provided LDBS blockstore. If this driver is not for a disk image file, there's no need to implement this function.

## 10 Writing new drivers (derived from LDBS)

The technique when creating an LDBS-based driver is similar to a standalone driver, with a few important differences. A good example file to look for this is the QRST driver.

An LDBS-based driver will need to make extensive use of the functions in `lqps.h`. Currently the best documentation for these functions is in the comments of `lqps.h` itself.

### 10.1 The driver header

As for a standalone driver, create a header, basing it on `example.h`. The first thing in the header (after the LGPL banner) is:

```
lqps_t struct
{
    LDBSDISK_D2K_DBIAEB prt_supt;
    char *prt_filename;
    /* The following variables hold state when saving and are only
```

```

    * used within drv_close() *\
    size_t drv_ioctl
    unsigned long drv_read
    unsigned long drv_write
    DRV_IOCTL_IOCTL

```

The major difference here is that the first member of the `struct` is a `DRV_IOCTL_IOCTL` rather than a plain `DRV_IOCTL_IOCTL`.

## 10.2 The driver source file

Secondly, create a `.c` file for your driver. Again, it's probably easiest to base this on `drivers.c`. At the start of this file, create a `DRV_CLASS` structure, `bus`:

```

DRV_IOCTL_IOCTL bus =
    sizeof(DRV_IOCTL_IOCTL)
    drv_ioctl_ioctl,
    DRV_IOCTL_IOCTL,
    "Driver Name",
    drv_read,
    drv_write,
    drv_ioctl_ioctl

```

The first four entries in this structure are:

The size of your driver's instance data

The driver's superclass. This needs to be set to `drv_ioctl_ioctl`, rather than `NULL` as it would be in a standalone driver.

The driver's name (as passed to `drv_ioctl_ioctl` / `drv_ioctl_ioctl`)

The driver's description string.

The remainder of the structure is composed of function pointers but you should only need to provide the first three pointers (`*_open`, `*_create` and `_close`).

Once you have created this structure, edit:

`drivers.h`. Add a declaration for your `DRV_CLASS` structure such as

```

extern DRV_IOCTL_IOCTL drv_ioctl_ioctl

```

`drivers.inc`. Insert a reference to your structure (eg: `"drv_ioctl_ioctl"`) in the list. Note that order is important the comments in `drivers.inc` describe how to decide where things go.

Edit `lib/Makefile.am`. Near the top of this file is a list of `drivers` and their header files just add your `.c` and `.h` to this list.

If your driver depends on certain system headers (as all `lib` drivers do) then you will need to add checks for these to `configure.in` and `lib/irvi.h` then run `autoconf` to rebuild the configure script.

The function pointers in the DRV\_CLASS structure are defined in drv.h. The first parameter to all of them ("self") is declared as a pointer to DRV\_DRIVER. In fact, it is a pointer to the first member of your instance data structure. You must cast the pointer to the correct type:

```
\* 2nd parameter: is type member for our driver; \
it (self -> drv = drv) let us DSK_ERR_BADBLOCK
return = (DRV_DSK_DRIVER *)self
```

and you're in business.

### 10.3 Driver functions

#### 10.3.1 dc\_open

```
void dc_open(struct drv *drv, const char *filename)
```

Open a disc image and load it into an LDBS blockstore. Enter with:

"self" points to the instance data for this disc image (see above) it will have been initialised to zeroes using memset().

"filename" is the name of the image to open.

Return:

DSK\_ERR\_OK: The driver has successfully opened the image.

DSK\_ERR\_NOTMEM: The driver cannot handle this image. Other drivers should be allowed to try to use it.

other: The driver cannot handle this image. No other drivers should be tried (eg: the image was recognised by this driver, but is corrupt).

Once you have established that your driver can open and pass the image it was passed, it should initialise the LDBS blockstore in the superclass using ldbst\_init():

```
void ldbst_init(struct ldbst *ldbst, const char *filename, const char *block)
```

In this case, the first parameter should point to the blockstore in the superclass. The other two should be NULL and LDBS\_DSK\_TYPE respectively:

```
ldbst_init(drv->ldbst, filename, NULL, LDBS_DSK_TYPE)
```

You then need to read in the entire disc image and save it in the blockstore using LDBS functions:

For each track, use ldbst\_track\_header() to create a track header structure.

For each sector in that track, use ldbst\_sector\_id() to generate its block ID, then ldbst\_block\_id() to add it to the store. Record the block ID in that sector's entry in the track header.

Once all the sectors have been added, use `ipps_block_finalize()` to add the completed track header.

If the file has a comment, record it using `ipps_block_comment()`, rather than `ipps_set_comment()`. If your disc image has a fixed geometry you should convert it to `DISK_GEOMETRY` structure and record it with `ipps_block_geometry()`. A few disc image formats contain a CP/M Disk Parameter block if yours is one you should add that with `ipps_block_dpb()`. Once the blockstore is completely populated, end with

```
return ipps_block_finalize();
```

### 10.3.2 dc\_creat

```
ipps_err_t (*dc_create)(DISK_GEOMETRY *geom, const char *filename)
```

This should check that the target file can be created, and hold of either its filename or file handle. It should then behave as 'open', except that you don't do anything with the blockstore created by `ipps_block_create()` before calling `ipps_block_finalize()`.

### 10.3.3 dc\_close

```
ipps_err_t (*dc_close)(DISK_GEOMETRY *geom)
```

Close the disc image. If it has been changed, you need to write the contents of the blockstore to a new file, overwriting anything that was there already.

The first thing to do (after basic sanity checks) is to call `ipps_block_flush()` to ensure all pending buffers have been written to the blockstore. The next is to check if any changes need to be written back if `ipps_block_dirty()` is zero, there's nothing to write back, so you can just close the blockstore and return.

```
return ipps_block_flush_and_write_back(geom);
```

Assuming that changes do need to be written back, you now need to reverse the conversion done by your 'open' method, and write the blockstore onto your disc image format. Some functions which may be helpful here are:

`ipps_max_cyl_heads()` will return the number of cylinders and heads necessary to contain this disc image. For example, a 720k DOS disc image would return 80 cylinders, 2 heads.

`ipps_get_ranges()` will analyse the blockstore and return the maximum and minimum values for cylinder, head, sector, sectors per track and sector size.

`ipps_iter_tracks()` will iterate over all tracks in the blockstore and, for each track, call a callback function you provide. It can return tracks in `SIDES_ALT`, `SIDES_OUTOUT` or `SIDES_OUTBACK` order if you want to process them in a different order you'll need to write your own iteration function. In that case, use `ipps_max_cyl_heads()` to get the range, and then `ipps_get_ranges()` for each track you want to process.

`ipps_iter_sectors()` works on the same principle, but the callback is run for each sector in each track. Note that sectors will be returned in the order they are listed in the track header, which may well be different from their numerical order.

`disk_get_block()` will load a sector using the block ID given for it in the track header.

`disk_get_comment()` will return the comment (if any). Similarly `disk_get_dpb()` retrieves the CP/M DPB (if one was written) and `disk_get_geom()` returns the DSK\_GEOMETRY last used to format a track. Note that the geometry recorded is advisory if it conflicts with the values returned by `disk_get_size()`, the values returned by `disk_get_size()` are going to be the accurate ones.

`disk_get_data()` will return a memory buffer containing the data from all the sectors in a particular track, in sector ID order.

`disk_close()` will be the last call your driver will receive for a given disc image file, and it should free any resources it is using. Whether it returns `DISK_ERR_OK` or an error, this disc image will not be used again.

## 11 Adding new compression methods

Adding a new compression method is very similar to adding a driver, though you only have to implement four functions.

To add a new driver, you create a new `COMPRESS_CLASS` structure and add it to various files.

### 11.1 Driver header

This is done as for disc drivers. If you don't need any extra variables (for example, gzip and bzip2 compression don't) then you don't have to define a new structure type - see `lib/compgz.h` for an example.

### 11.2 Driver implementation

Secondly, create a `.c` file for your driver. It's probably easiest to base this on `lib/compgz.c`. At the start of this file, create a `COMPRESS_CLASS` structure such as:

```
COMPRESS_CLASS gzf =
{
    sizeof(COMPRESS_DATA),
    0,
    GZIP (gzip compression),
    gzf_open \* gzf_open *\
    gzf_close \* gzf_close *\
    gzf_read \* gzf_read *\
    gzf_write \* gzf_write *\
}
```

The first three entries in this structure are:

The size of your driver's instance data. The GZip driver has instance data and so just uses `COMPRESS_DATA`. If it had extra data these would be a structure called `GZ_COMPRESS_DATA`, so the size here would be `sizeof(GZ_COMPRESS_DATA)`.

The driver's name (as passed to `dsk_open()` / `dsk_create()`)

The driver's description string.

The remainder of the structure is composed of function pointers. The types of these are given in drv.h. You must implement all four.

Once you have created this structure, edit:

comp.h. Include your header.

compress.inc. Insert a reference to your structure (eg: "cc\_myzip,") in the list.

Note that order is important.

Edit "lib/Makefile.am". At the bottom of this file is a list of files and their header files just add your .c and .h to this list.

If your driver depends on certain system headers (eg, the gz library depends on zlib.h) then you will need to add checks for these to "configure" and "lib/compi.h" then run "autoconf" to rebuild the configure script.

The function pointers in the COMPRESS\_CLASS structure are described in lib/compress.h. The first parameter to all of them ("self") is declared as a pointer to COMPRESS\_DATA. In fact, it is a pointer to the first member of your instance structure. Just cast the pointer to the correct type:

```
\* Sanity check: Is this meant for our driver? *\
if (self->cd_filename == NULL) return DSK_ERR_BADFILE
return (ZOO_COMPRESS_DATA *)self
```

and you're in business.

### 11.3 Compression functions

#### 11.3.1 cc\_open

```
int cc_open(const char *filename, COMPRESS_DATA *self)
```

Attempt to decompress a compressed file.

"self" points to the instance data for this disc image.

self-cd\_filename is the filename of the file to decompress.

Return:

DSK\_ERR\_OK: The file has been decompressed.

DSK\_ERR\_NOTMEM: The file is not compressed using this driver's method.

other: The file does not belong to this driver, but it is corrupt or some other error occurred.

Two helper functions may be useful when you are writing cc\_open

```
int cc_open(const char *filename, COMPRESS_DATA *self, FILE **fp)
```

Open the file whose name is given at filename. If successful, fp will be the opened stream. If not, it will be NULL. If the file is opened read-only, sets self-cd\_readonly to 1.

```
int cc_mktemp(const char *filename, COMPRESS_DATA *self, FILE **fp)
```

Create a temporary file and store its name at filename. You should use this to create the file that you decompress into.

### 11.3.2 cc\_creat

```
q2k6ll t (*cc6l69t)(COMBEE22 DVLV *c9)
```

Warn the compression engine that a disc image file is being ~~at~~, and when closed it will be compressed. The filename is stored at ~~26ll-c9-ll69996~~. Normally this just returns DSK\_ERR\_OK.

### 11.3.3 cc\_commit

```
q2k6ll t (*cc6999t)(COMBEE22 DVLV *c9)
```

Compress an uncompressed file. ~~26ll-c9-ll69996~~ is the name of the file to compress. ~~26ll-c9-ll69996~~ is the name of the output file.

### 11.3.4 cc\_abort

```
q2k6ll t (*cc999t)(COMBEE22 DVLV *c9)
```

This is used if a file was decompressed and it's now being ~~cl~~ without having been changed. There is therefore no need to compress it again. ~~Th~~ normally just returns DSK\_ERR\_OK.

## 12 Adding new remote transports.

Adding a new remote transport is also very similar to adding ~~driver~~.

To add a new driver, you create a new REMOTE\_CLASS structure ~~and~~ add it to various files.

### 12.1 Driver header

This is done as for disc drivers. Create a structure based on REMOTE\_DATA to hold your class's data see lib/rpctios.h and lib/rpcfork.h ~~for~~ examples.

### 12.2 Driver implementation

Create a .c file for your driver. It's probably easiest to ~~bt~~ this on lib/rpcfork.c. At the start of this file, create a REMOTE\_CLASS structure, such as:

```
REMOTE_CLASS rbc_fork =
    sizeof(ROBK_REMOTE_DVLA),
    fork,
    UNIX client name fork,
    fork_oben, \* oben *\
    fork_cj026, \* cj026 *\
    fork_c9ll, \* b6ll699 B6C *\
```

The first three entries in this structure are:

The size of your driver's instance data `sizeof(your_REMOTE_DATA)` structure.

The driver's name. If the filename passed to `LibDsk` begins with this name followed by a colon, then it's assumed to be using your driver.

The driver's description string.

The remainder of the structure is composed of function pointers. The types of these are given in `lib/remote.h`. You must implement all three.

Once you have created this structure, edit:

`lib/remall.h`. Include your header.

`lib/remote.inc`. Insert a reference to your structure (`@pc_fork;`) in the list.

The drivers will be tested in the order in which they appear in the file.

Edit "`lib/Makefile.am`". At the bottom of this file is a list of drivers and their header files just add your `.c` and `.h` to this list.

If your driver depends on certain system headers (eg, `termios`), then you will need to add checks for these to "`configure.in`" and "`lib/drvi.h`" then run "`autoconf`" to rebuild the configure script.

The function pointers in the `REMOTE_CLASS` structure are defined in `lib/compress.h`. The first parameter to all of them ("`pDriver`") is declared as `DSK_DRIVER` you can extract a pointer to your instance data using the `dr_remote` member like this:

```
\* 2am!t\ cpeck2 *\
26H = (EOBK REMOTE_DVIV *)bDl!aGL-ql-LEmo06
H (26H == N000 26H-20p6L-1q-cjss = 1bC 10Lk)
160000 D2K-EBK BVDP1K
```

## 12.3 Remote communication functions

### 12.3.1 `rc_open`

```
26K-6L-t (*LC-ob60) (D2K-EBK bDl!aGL' 002t cpaL *pam6' cpaL *pam60t)
```

Connect to a remote server.

`pDriver` points to a `DSK_DRIVER` containing the pointer to instance data.

`name` is the filename as passed to `LibDsk`, starting with "driver:" and containing any connection parameters needed.

`nameout` is an output buffer with enough space to hold a string the same length as the input filename. If you are returning `DSK_ERR_OK`, it is set to the input filename minus any options this driver has used. For example, the "serial" driver, given a filename like "serial:/dev/ttyS1,2400-cts,example.ufi,raw" would extract its own options and return "example.ufi,raw" here.

Return:

`DSK_ERR_OK`: Connection established.

`DSK_ERR_NOTME`: The filename passed is not recognised by this driver.

other: An error such as out-of-memory occurred.

### 12.3.2 rc\_close

q2k-6ll-t (\*LC-GJ026)(D2K-BDKIAEK bDl!A6L)

Close the connection to the remote server.

### 12.3.3 rc\_call

q2k-6ll-t (\*LC-G9II)(D2K-BDKIAEK bDl!A6L' nuzigued cpaL \*inbu' in' inb-]en' nuzigued cpaL \*o

Perform a remote procedure call to the server.

**input** is the packet LibDsk wants to send.

**inp\_len** is the number of bytes in the packet.

**output** is a buffer for the result packet.

**\*out\_len** (on entry) is the size of the result buffer.

**\*out\_len** (on return) is the number of bytes that were populated in **output** buffer.

In general, this call will wrap the input in whatever framing bytes are necessary (usually including the packet length, since packets do not contain their own length), send the packet over the wire, wait for a response, and unpack the response into 'output'. Return DSK\_ERR\_TIMEOUT if the connection timed out (the 'real' driver waits 30 seconds) and DSK\_ERR\_ABORT if the user deliberately broke connection.

## A The CopyQM File Format

### A.1 Introduction

This section describes the file format of files created by CopyQM. A lot of the information has been extracted by looking at hex-dumps of the files there might be some errors in the description.

### A.2 Header

The CopyQM files consist of a header, an optional comment (indicated by the header) followed by the tracks of the image encoded with a run length encoding scheme. The header is 133 bytes long, see table. It always starts with 0x03, 0x51, 0x14, which can be used for auto-detection of the image. All numbers have little-endian byte ordering. When all bytes in the header are added together in a byte, the result should be zero.

Offset	Size	Comment
0x00	1	Always 0x43 ('C')
0x01	1	Always 0x51 ('Q')
0x02	1	Always 0x14
0x03	2	Sector size (from here to 0x1B inclusive is a DOS BPB)
0x05	1	Sectors per cluster
0x06	2	Number of reserved sectors
0x08	1	FAT copies
0x09	2	Number of root directory entries
0x0b	2	Total number of sectors
0x0d	1	Media byte
0x0e	2	Number of sectors per FAT
0x10	2	Number of sectors per track
0x12	2	Number of heads
0x14	4	Number of hidden sectors
0x18	4	Total sectors if 65535 (should always be 0 on a floppy image)
0x1c	60?	Description of media (e.g. "720K Double-Sided")
0x58	1	Type of image. 0=DOS, 1=blind, 2=HFS
0x59	1	Density. 0=DD, 1=HD, 2=ED
0x5a	1	Number of tracks used on image
0x5b	1	Total number of tracks for image
0x5c	4	CRC for the used, unpacked tracks
0x60	11	Volume label (DOS/HFS)
0x6b	2	Creation time
0x6d	2	Creation date
0x6f	2	Length of image comment
0x71	1	Number of first sector - 1
0x74	1	Interleave. (0 for older versions of CopyQM)
0x75	1	Skew. Normally 0. Negative number for alt. sides
0x84	1	Header checksum byte

### A.3 CRC

The CRC is calculated for the unpacked data for all tracks ~~that~~ used in the image. The CRC value is initialized with 0 and then updated using CRC 32 polynomial 0x104C11DB7, bit reverse algorithm. Due to a feature in CopyM (8 bit register as an index into a 1024 byte table) all bytes must have their top bits removed before added to the CRC.

### A.4 Image comment

The image comment follows the header. It has a variable size ~~and~~ in the header. The image comment can contain 0-bytes.

### A.5 Image data

The image data is run length encoded. Each run is preceded by ~~6~~ bit length. If the length is negative, the byte after the length is repeated ~~length~~ times. If the length is positive, it is followed by ~~length~~ bytes of unencoded data. It seems like a new run

of repeating or differing data is always started at each new track. Older versions of CopyQM always alternates between runs of differing data and repeating data, even if the length of one of them is zero.

## B DQK Files

A DQK file is a .DSK file compressed using Richard Greenlaw's ~~sq~~sq file format (originally from CP/M as SQ.COM, and later built in to ~~NSWDOS~~NSWDOS versions also exist for DOS and UNIX). SQ was used in preference to more efficient compressors such as gzip because it can be readily decoded on 8-bit and ~~16-bit~~16-bit computers.

The original reason for DQK files was software distribution. A disc image of a 180k disc won't fit on a 180k disc, owing to various overheads. However, the compressed DQK version may fit onto such a disc, and leave room for a tool to write the DQK back out as well.

Such a tool has been included in the "dskwrite" directory ~~in this~~in this distribution. It contains the following files:

dskwrite.com: Program to write .DSK or .DQK files out to a ~~hard~~disc. The .COM file works on PCs, Amstrad PCWs and Sinclair Spectrum.3s

dskwrite.txt: Documentation for dskwrite.

dskwrite.z80: Z80 source for the CP/M version.

dskwrite.asm: 8086 source for the DOS version.

dskwrsea.com: The dskwrite distribution file - a self-extracting archive. It will self-extract under CP/M or DOS.

Note that the files in the "dskwrite" directory are not GPLed. They are public domain. You may do whatsoever you please with them.

LibDsk has been given .DQK support (use the "dsk" driver with "z" compression) so that .DQK files don't have to be created and compressed in a separate process.

## C LibDsk with cpmtools

cpmtools v1.9 and later <http://www.moria.de/michael/qpmtools/> can be configured to use LibDsk for all disc access, thus allowing CP/M discs and emulator disc images to be read and written.

cpmtools v2.19 allow a cpmtools disk definition to be associated with a LibDsk geometry, by adding a "libdsk:format" entry to the disk definition. For example, this entry uses "libdsk:format ibm1440" so that the disk ~~is~~is accessed using the 'ibm1440' geometry rather than the default 'pcw1440'.

```
q1zkrqel cbwzge-1v4t6st
z6c16v 215
r1zcr2 100
z6c1r1 18
p10c1z1x6 4000
wzqxq1r 520
zkr6w 1
```

```

rootflk 3
os 3
diskdef:format ipm1440
end

```

The myz80 and nanowasp drivers use a fixed disk format here at diskdefs entries which can be used to read them:

```

diskdef myz80
sector 1054
tracks 44
sector 138
blocks 4096
maxdir 1054
rpm 1
rootflk 0
os 3
end
diskdef nanowasp
sector 215
tracks 80
sector 10
blocks 5048
maxdir 138
rpm 1
os 33
end

```

In the old diskdefs format with one line per entry, these are:

```

myz80 1054 44 138 4096 1054 1 0 3
nanowasp 215 80 10 5048 138 1 3 33

```

## D DSK / EDSK recording mode extension

This extension was proposed by me on the comp.sys.sinclair and comp.sys.amstrad.8bit newsgroups on 10 January 2004. It was subsequently released in ANNE 2.1.4 and added to the formal EDSK format definition at <http://andheran.aiind.upv.es/amstrad/docs/extdsk.html>

DSK/EDSK originate on the Amstrad CPC, which ordinarily uses all its diskettes in MFM recording mode and at the Double Density rate. However, ANNE emulates the PcW16, which also supports the High Density rate and the system software depends on DD discs not being readable at the HD rate.

The extension gives meanings to two unused bytes of the DSK/EDSK “Track-Info” block:

Byte 12h: Data rate.

0 Unknown

1 Single or Double Density (180k, 720k, etc.)

**2 High Density (1.2M, 1.4M, etc.)**

**3 Extended Density (2.8M)**

**Byte 13h: Recording mode.**

**0 Unknown**

**1 FM**

**2 MFM**

**Existing files should have zeroes in these bytes hence the use of 0 for Unknown. LibDsk will guess the values in if the ones in the file are zero.**