LibDsk v1.1.0

John Elliott

13th January 2003

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 drives.

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

Contents

1 Introduction

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;
- 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;
- The floppy drive under Linux;
- The floppy drive under Windows. Windows support is a complicated subject - see section 5 below.
- The floppy drive under DOS.

LibDsk also supports compressed disc images in the following formats:

- Squeeze (Huffman coded)
- GZip (Deflate)
- BZip2 (Burrows-Wheeler; support is read-only)

1.1 What's new?

For full details, see the file ChangeLog.

- The parameter order of the example utilities has been made less rigid.
- The Windows version of LibDsk contains a miniature RPC system to allow 32-bit programs to use the more capable 16-bit drivers. See section 5.5 for more information.
- A driver has been added for the disc image files used by the NanoWasp emulator.
- LibDsk compiles under DOS. Makefiles are provided for DJGPP and Pacific C.
- LibDsk compiles under 16-bit Windows. Project files are provided for the Borland and Microsoft compilers.
- A bug fix in the Posix and MYZ80 drivers, in which files were extended with zeroes (Unix) or random data (Win32).
- IMPORTANT CHANGES: Version 0.9.0 breaks backward compatibility in a number of ways. Most importantly, you must replace "DSK_DRIVER *" with "DSK_PDRIVER " throughout your programs.
- Disc images can be compressed and decompressed on the fly.

1.2 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. For a single-sided disc, these are the same; for a double-sided disc, there are twice 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=/dev/fd0 of=image`". On systems other than Linux, DOS or Windows, this is also used to access the host system's floppy drive.

"floppy" : Host system's floppy drive (under Linux, DOS or Windows).

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

"cfi" : Compressed floppy image, as produced by FDCOPY.COM under DOS. Its format is described in cfi.html.

"nanowasp" : Disc image in the 400k Microbee format used by the NanoWasp emulator. This is similar to "raw", but the tracks are stored in a 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.

3 Architecture

LibDsk is composed of a fixed core (files named `dsk*.c`) and a number of drivers (files named `drv*.c`). When you open an image or a drive (using `dsk_open()` or `dsk_creat()`) then a driver is chosen. This driver is then used until it's closed (`dsk_close()`).

Each driver is identified by a name. To get a list of available drivers, use `dsk_type_enum()`. To get the driver that is being used by an open DSK image, use `dsk_drvname()` or `dsk_drvdesc()`.

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 (79,1,9).

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 `DSK_GEOMETRY` structure to convert to C/H/S. `DSK_GEOMETRY` also contains information such as the sector size and data rate used to access a given 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 `DSK_GEOMETRY` structure, either:

- call `dsk_getgeom()` to try and detect it from the disc; or
- call `dg_stdformat()` to select one of the "standard" formats that LibDsk knows about; or
- call `dg_dosgeom()` / `dg_cpm86geom()` / `dg_pcwgeom()` to initialise it from a copy of a DOS / CP/M86 / PCW boot sector; or
- Set all the members manually.

3.1.1 DSK_GEOMETRY in detail

```
typedef struct
{
```

`dsk_sides_t dg_sidedness;` /* This describes the logical sequence of tracks on the disc - the order in which their host system reads them. This will only be used if `dg_heads` 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_ALT` 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_OUTBACK` 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 0 Head 1 is the last). This layout is used by Freek Heite's 144FEAT driver (for CP/M-86 on the PC) but I have not seen it elsewhere.

`SIDES_OUTOUT` 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.

*/

`dsk_pcyl_t dg_cylinders;` /* The number of cylinders this disc has. Usually 40 or 80. */

`dsk_phead_t dg_heads;` /* The number of heads (sides) the disc has. Usually 1 or 2. */

`dsk_psect_t dg_sectors;` /* The number of sectors per track. */

`dsk_psect_t dg_secbase;` /* 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. */

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

`dsk_rate_t dg_datarate;` /* Data rate. This will be one of:

`RATE_HD`	High-density disc (1.4Mb or 1.2Mb)
`RATE_DD`	Double-density disc in 1.2Mb drive (ie, 360k disc in 1.2Mb drive)
`RATE_SD`	Double-density disc in 1.4Mb or 720k drive
`RATE_ED`	Extra-density disc (2.8Mb) */

`dsk_gap_t dg_rwgap;` /* Read/write gap length */

`dsk_gap_t dg_fmtgap;` /* Format gap length */

`int dg_fm;` /* Set to nonzero to use FM (single density) recording mode. Not all PC floppy controllers support this mode; the National Semiconductor PC87306 and the 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. The Windows / DOS floppy drivers do not support FM recording. */

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

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

```
} DSK_GEOMETRY;
```

4 LibDsk Function Reference

4.1 dsk_open: Open an existing disc image

```
dsk_err_t dsk_open(DSK_PDRIVER *self, const char *file-
name, const char *type, const char *compress)
```

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 Win32, "A:" and "B:" refer to the two 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_type_enum()` below.
- "compress" is NULL to auto-detect compressed files, or the name of a LibDsk compression scheme. See `dsk_comp_enum()`.

Returns: A `dsk_err_t`, which will be 0 (`DSK_ERR_OK`) if successful, or a negative integer if failed. See `dsk_strerror()`. The error `DSK_ERR_NOTME` 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 A).

"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 bzlib support.

4.2 dsk_creat: Create a new disc image

```
dsk_err_t dsk_creat(DSK_PDRIVER *self, const char *file-
name, 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 for `dsk_open()`, except that "type" cannot be NULL (it must specify the type of disc image 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(DSK_PDRIVER *self)
```

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.

4.4 dsk_pread, dsk_lread : Read a sector

```
dsk_err_t dsk_pread(DSK_PDRIVER self, const DSK_GEOMETRY *geom, void *buf,
der, dsk_phead_t head, dsk_psect_t sector)
dsk_err_t dsk_lread(DSK_PDRIVER self, const DSK_GEOMETRY *geom, void *buf,
tor)
```

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" (`dsk_pread`) or "sector" (`dsk_lread`) 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.

4.5 dsk_pwrite, dsk_lwrite: Write a sector

```
dsk_err_t dsk_pwrite(DSK_PDRIVER self, const DSK_GEOMETRY *geom, const voic
der, dsk_phead_t head, dsk_psect_t sector)
dsk_err_t dsk_lwrite(DSK_PDRIVER self, const DSK_GEOMETRY *geom, const voic
tor)
```

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.

4.6 dsk_pcheck, dsk_lcheck: Verify sectors on disc against memory

```
dsk_err_t dsk_pcheck(DSK_PDRIVER self, const DSK_GEOMETRY *geom, const voic
der, dsk_phead_t head, dsk_psect_t sector)
dsk_err_t dsk_lcheck(DSK_PDRIVER self, const DSK_GEOMETRY *geom, const voic
tor)
```

As `dsk_pread` / `dsk_lread`, but rather than reading their buffers from disc, they compare the contents of their buffers with the data already on the 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 the driver cannot read sectors, DSK_ERR_NOTIMPL will be returned.

4.7 dsk_pformat, dsk_lformat: Format a disc track

```
dsk_err_t dsk_pformat(DSK_PDRIVER self, DSK_GEOMETRY *geom, dsk_pcyl_t cyli
der, dsk_phead_t head, const DSK_FORMAT *for-
mat, unsigned char filler)
dsk_err_t dsk_lformat(DSK_PDRIVER self, DSK_GEOMETRY *geom, dsk_ltrack_t tr
mat, unsigned char filler)
```

Enter with:

- “self” is a handle to an open drive / image file.
- “geom” points to the geometry for the drive. The formatter may modify this if (for example) it’s asked to format track 41 of a 40-track drive.
- “cylinder” / “head” (`dsk_pformat`) or “track” (`dsk_lformat`) give the location of the track to format.

- "format" should be an array of (`geom->dg_sectors`) DSK_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, DSK_ERR_NOTIMPL will be returned.

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

4.8 dsk_apform, dsk_alform: Automatic format

```
dsk_err_t dsk_apform(DSK_PDRIVER self, const DSK_GEOMETRY *geom, dsk_pcyl_t
der, dsk_phead_t head, unsigned char filler)
dsk_err_t dsk_alform(DSK_PDRIVER self, const DSK_GEOMETRY *geom, dsk_ltrack
signed char filler)
```

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

4.9 dsk_psecid, dsk_lsecid: Read a sector ID.

```
dsk_err_t dsk_psecid(DSK_PDRIVER self, const DSK_GEOMETRY *geom, dsk_pcyl_t
der, dsk_phead_t head, DSK_FORMAT *result)
dsk_err_t dsk_lsecid(DSK_PDRIVER self, const DSK_GEOMETRY *geom, dsk_ltrack
sult)
```

Read a sector ID from the given track. This can be used to probe for 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" (`dsk_psecid`) or "track" (`dsk_lsecid`) give the location of the track to read the sector from.
- "result" points to an uninitialised `DSK_FORMAT` structure.

On return:

- If successful, the buffer at "result" will be initialised with the sector header found, and DSK_ERR_OK will be returned.
- If the driver cannot provide this functionality (for example, the Win32 driver under NT), DSK_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 / PCW discs and CPC discs. However it will not be usable for other purposes.

4.10 dsk_xread, dsk_xwrite: Low-level reading and writing

```
dsk_err_t dsk_xread(DSK_PDRIVER self, const DSK_GEOMETRY *geom, void *buf,
der, dsk_phead_t head, dsk_pcyl_t cyl_expected, dsk_phead_t head_expected,
tor, size_t sector_len, int *deleted);
dsk_err_t dsk_xwrite(DSK_PDRIVER self, const DSK_GEOMETRY *geom, const voic
der, dsk_phead_t head, dsk_pcyl_t cyl_expected, dsk_phead_t head_expected,
tor, size_t sector_len, int deleted);
```

dsk_xread() and dsk_xwrite() are extended versions of dsk_pread() and dsk_pwrite(). 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 “cylinder” 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 and the Linux floppy driver. Other drivers will return DSK_ERR_NOTIMPL. Unless you are emulating a floppy controller, or you need to read discs that contain deleted data or misnumbered sectors, it should not be necessary to call these functions.

4.10.1 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 `geom->dg_noskip` 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

4.11 dsk_ltread, dsk_ptread, dsk_xtread

```
dsk_err_t dsk_ltread(DSK_PDRIVER self, const DSK_GEOMETRY *geom, void *buf,
dsk_err_t dsk_ptread(DSK_PDRIVER self, const DSK_GEOMETRY *geom, void *buf,
der, dsk_phead_t head)
dsk_err_t dsk_xtread(DSK_PDRIVER self, const DSK_GEOMETRY *geom, void *buf,
der, dsk_phead_t head, dsk_pcyl_t cyl_expected, dsk_phead_t head_expected)
```

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, LibDsk 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_ptread`, `dsk_xtread`) or "track" (`dsk_ltread`) give the location of the track to read.
- (`dsk_xtread`) "cyl_expected" and "head_expected" are used as the values to search for in the sector headers.

Returns:

- If successful, DSK_ERR_OK. Otherwise, a negative DSK_ERR_* value.
- (`dsk_xtread()` only) If the driver does not support extended sector reads/writes, then DSK_ERR_NOTIMPL will be returned.

4.12 dsk_lseek, dsk_pseek

```
dsk_err_t dsk_lseek(DSK_PDRIVER self, const DSK_GEOMETRY *geom, dsk_ltrack_
dsk_err_t dsk_pseek(DSK_PDRIVER self, const DSK_GEOMETRY *geom, dsk_pcyl_t
der, dsk_phead_t head)
```

Seek to a given cylinder. Only the CPCEMU driver and the Linux 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.

4.13 dsk_drive_status

```
dsk_err_t dsk_drive_status(DSK_PDRIVER self, const DSK_GEOMETRY *geom, dsk_
signed char *result)
```

Get the drive's status (ready, read-only etc.). The byte "result" 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.

4.14 dsk_getgeom: Guess disc geometry

```
dsk_err_t dsk_getgeom(DSK_PDRIVER self, DSK_GEOMETRY *geom)
```

This attempts to determine the geometry of a disc (number of cylinders, tracks, sectors etc.) by loading the boot sector. It understands DOS, 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.

4.15 dg_*geom : Initialise disc geometry from boot sector

```
dsk_err_t dg_dosgeom(DSK_GEOMETRY *self, const un-
signed char *bootsect)
dsk_err_t dg_pcwgeom(DSK_GEOMETRY *self, const un-
signed char *bootsect)
dsk_err_t dg_cpm86geom(DSK_GEOMETRY *self, const un-
signed char *bootsect)
```

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

- "self" is the structure to initialise;
- "bootsect" is the boot sector to initialise the structure from.

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

4.16 dg_stdformat : Initialise disc geometry from a standard LibDsk format.

```
dsk_err_t dg_stdformat(DSK_GEOMETRY *self, dsk_format_t for-
matid, dsk_cchar_t *fname, dsk_cchar_t *fdesc)
```

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 "out-and-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

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/dskform.c`).

If the "fdesc" is not NULL, it will be pointed at a description string 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/formnames.c` for example code that does this.

4.17 dsk_*_forcehead: Override disc head

```
dsk_err_t dsk_set_forcehead(DSK_PDRIVER self, int force)
dsk_err_t dsk_get_forcehead(DSK_PDRIVER self, int *force)
```

(This option is only effective for the Linux floppy driver)

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 PCW / CPC / Spectrum+3 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.

4.18 dsk_type_enum

```
dsk_err_t dsk_type_enum(int index, char **drv-
name)
```

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

4.19 dsk_comp_enum

```
dsk_err_t dsk_comp_enum(int index, char **comp-
name)
```

As `dsk_type_enum()`, but lists supported compression schemes.

4.20 dsk_drvname, dsk_drvdesc

```
const char *dsk_drvname(DSK_PDRIVER self)
const char *dsk_drvdesc(DSK_PDRIVER self)
```

Returns the driver name (eg: “myz80”) or description (eg “MYZ80 hard drive driver”) for an open disc image.

4.21 dsk_compname, dsk_compdesc

```
const char *dsk_compname(DSK_PDRIVER self);
const char *dsk_compdesc(DSK_PDRIVER self);
```

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

4.22 dg_ps2ls, dg_ls2ps, dg_pt2lt, dg_lt2pt

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

```
dsk_err_t dg_ps2ls(const DSK_GEOMETRY *self, dsk_pcyl_t cyl, dsk_phead_t he
ical)
```

Converts physical C/H/S to logical sector.

```
dsk_err_t dg_ls2ps(const DSK_GEOMETRY *self, dsk_lsect_t log-
i-
cal, dsk_pcyl_t *cyl, dsk_phead_t *head, dsk_psect_t *sec)
```

Converts logical sector to physical C/H/S.

```
dsk_err_t dg_pt2lt(const DSK_GEOMETRY *self, dsk_pcyl_t cyl, dsk_phead_t he
ical)
```

Converts physical C/H to logical track.

```
dsk_err_t dg_lt2pt(const DSK_GEOMETRY *self, dsk_ltrack_t log-
ical, dsk_pcyl_t *cyl, dsk_phead_t *head)
```

Converts logical track to physical C/H.

4.23 dsk_strerror: Convert error code to string

```
char *dsk_strerror(dsk_err_t err)
```

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

4.24 dsk_get_psh

```
unsigned char dsk_get_psh(size_t sector_size)
```

Converts a sector size into the sector shift used by the uPD765A controller (eg: 128 -> 0, 256 -> 1, 512 -> 2 etc.) You should not need to use this. The reverse operation is: sectorsize = (128 < < psh).

4.25 Structure: DSK_FORMAT

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

fmt_cylinder: Cylinder number.

fmt_head: Head number.

fmt_sector: Sector number.

fmt_secsize: Sector size in bytes.

4.26 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 `dg_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 `dsk_open()` / `dsk_creat()`.

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: `malloc()` failed to allocate memory.

DSK_ERR_NOTIMPL: Function is not implemented (eg, this driver doesn't support `dsk_xread()`).

DSK_ERR_MISMATCH: In `dsk_lcheck()` / `dsk_pcheck()`, 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_UNKNOWN: Unknown error

4.27 Miscellaneous

LIBDSK_VERSION is a macro, defined as a string containing the library version - eg "0.9.0"

5 LibDsk under Windows

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

As with so many other aspects of Windows, direct access to the floppy drive is a case of "write once - debug everywhere"[1]. 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.

5.1 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 can, for example, read Acorn ADFS floppies.

5.2 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, which the 32-bit version cannot. Unfortunately, 32-bit programs can't link to the 16-bit version of LibDsk[2], but there is a workaround (described below) involving the use of LDSERVER.

5.3 Windows NT (NT 3.x, NT 4.x, 2000, XP)

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

5.4 General comments on programming floppy access for Windows

LibDsk has three independent drivers for accessing floppies under Windows. They are:

5.4.1 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.

[1]Originally said by Microsoft with respect to Java. Pot. Kettle. Black.

[2]And no, the Generic Thunk isn't good enough. I've tried it.

5.4.2 The Win32c driver.

This driver uses VWIN32 services to make INT 0x13-style calls under Windows9x. 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 isn't 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.

5.4.3 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 "raw" driver on "`\\.\A:`" , in practice there are a number of nasty subtleties relating to such things as memory alignment and file locking.

5.4.4 Other floppy APIs

Sydex produce a replacement floppy driver for 32-bit versions of Windows (SydexFDD) which removes most of the restrictions that Windows places on acceptable floppy formats. LibDsk does not support this.

5.5 LDSERVER

LDSERVER is a program that makes the 16-bit LibDsk DLL available to 32-bit programs. It does this by creating a mailslot ("`\\.\mailslot\LibDsk16`") 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.

5.5.1 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 NETAPI.LIB NETAPI.DLL
```

or the equivalent utility for your compiler.

5.5.2 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 disc access.

LDSERVER does not shut down automatically.

5.5.3 Important Security Warning

LDSERVER is a 16-bit program, written using APIs intended for use on a local area network. 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 attacker could use LDSERVER to overwrite important system files or read confidential documents.

If you have a firewall, then make sure that the NetBIOS ports 137, 138 and 139 are blocked. If you don't have a firewall, ***do not run LDSERVER while your computer is connected to the Internet!***

6 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.

6.1 The driver header

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

```
typedef struct
{
    DSK_DRIVER px_super;
    FILE *px_fp;
    int px_readonly;
    long px_filesize;
} POSIX_DSK_DRIVER;
```

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 I will come back to later.

6.2 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:

```
DRV_CLASS dc_posix =
{
    sizeof(POSIX_DSK_DRIVER),
    "raw",
    "Raw file driver",
    posix_open,
    posix_creat,
    posix_close
};
```

The first three entries in this structure are:

- The size of your driver's instance data;
- The driver's name (as passed to `dsk_open()` / `dsk_creat()`)
- 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 DRV_CLASS dc_myformat;
  ```

- drivers.inc. Insert a reference to your structure (eg: "`&dc_myformat,`") 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 floppy drivers do) 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 DRV_CLASS structure are described in drv.h. The first parameter to all of them ("self") is declared as a pointer to DSK_DRIVER. In fact, it is a pointer to the first member of your instance data structure. Just cast the pointer to the correct type:

```
/* San-
ity check: Is this meant for our driver? */
if (self-
>dr_class != &dc_posix) return DSK_ERR_BADPTR;
pxself = (POSIX_DSK_PDRIVER )self;
```

and you're in business.

6.3 Driver functions

6.3.1 dc_open

```
dsk_err_t (*dc_open )(DSK_PDRIVER self, const char *file-
name)
```

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).

6.3.2 dc_creat

```
dsk_err_t (*dc_creat)(DSK_PDRIVER self, const char *file-
name)
```

Attempt to create a new disc image. For the "floppy" drivers, 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.

6.3.3 dc_close

dsk_err_t (*dc_close)(DSK_PDRIVER 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.

6.3.4 dc_read

```
dsk_err_t (*dc_read)(DSK_PDRIVER self, const DSK_GEOMETRY *geom, void *buf,
der, dsk_phead_t head, dsk_psect_t sector)
```

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_pread().

6.3.5 dc_write

```
dsk_err_t (*dc_write)(DSK_PDRIVER self, const DSK_GEOMETRY *geom, const voi
der, dsk_phead_t head, dsk_psect_t sector)
```

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.

6.3.6 dc_format

```
dsk_err_t (*dc_format)(DSK_PDRIVER self, const DSK_GEOMETRY *geom, dsk_pcyl
der, dsk_phead_t head, const DSK_FORMAT *for-
mat, unsigned char filler)
```

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.

6.3.7 dc_getgeom

```
dsk_err_t (*dc_getgeom)(DSK_PDRIVER self, DSK_GEOMETRY *geom)
```

Get the disc geometry. Leave this function pointer as NULL unless your disc image does not allow a caller to use an arbitrary disc geometry.

The two 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 one, which has a single fixed geometry.

Return DSK_ERR_OK if successful; DSK_ERR_NOTME to fall back to the standard LibDsk geometry probe; other values to indicate failure.

6.3.8 dc_secid

```
dsk_err_t (*dc_secid)(DSK_PDRIVER self, const DSK_GEOMETRY *geom, dsk_pcyl_
der, dsk_phead_t head, DSK_FORMAT *result)
```

Read the ID of a random sector on a certain track/head, and put it in "result". This function is primarily 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.

6.3.9 dc_xseek

```
dsk_err_t (*dc_xseek)(DSK_PDRIVER self, const DSK_GEOMETRY *geom, dsk_pcyl_
der, dsk_phead_t head);
```

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 by itself.

6.3.10 dc_xread, dc_xwrite

```
dsk_err_t (*dc_xread)(DSK_PDRIVER self, const DSK_GEOMETRY *geom, void *buf
der, dsk_phead_t head, dsk_pcyl_t cyl_expected, dsk_phead_t head_expected,
tor, size_t bytes_to_write, int *deleted);
dsk_err_t (*dc_xwrite)(DSK_PDRIVER self, const DSK_GEOMETRY *geom, const vo
der, dsk_phead_t head, dsk_pcyl_t cyl_expected, dsk_phead_t head_expected,
tor, size_t bytes_to_read, int *deleted);
```

Read / write sector whose ID may not match its position on disc, or which is marked as deleted. Only implement this if your disc image emulates sector IDs or your floppy driver exposes this level of functionality. Currently it is only implemented in the Linux and CPCEMU drivers.

6.3.11 dc_status

```
dsk_err_t (*dc_status)(DSK_PDRIVER self, const DSK_GEOMETRY &geom, dsk_phea
signed char *result);
```

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

6.3.12 dc_tread

```
dsk_err_t (*dc_tread)(DSK_PDRIVER self, const DSK_GEOMETRY *geom, void *buf
der, dsk_phead_t head);
```

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_ptread().

6.3.13 dc_xtread

```
dsk_err_t (*dc_xread)(DSK_PDRIVER self, const DSK_GEOMETRY *geom, void *buf
der, dsk_phead_t head, dsk_pcyl_t cyl_expected, dsk_phead_t head_expected);
```

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.

7 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.

7.1 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 declare a new structure type - see lib/compgz.h for an example.

7.2 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 cc_gz =
{
    sizeof(COMPRESS_DATA),
    "gz",
    "Gzip (deflate compression)",
    gz_open, /* open */
    gz_creat, /* create new */
    gz_commit, /* commit */
    gz_abort /* abort */
};
```

The first three entries in this structure are:

- The size of your driver's instance data. The GZip driver has no instance data and so just uses COMPRESS_DATA. If it had extra data these would be in a struct called GZ_COMPRESS_DATA, so the size here would be sizeof(GZ_COMPRESS_DATA).
- The driver's name (as passed to dsk_open() / dsk_creat())
- 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 drivers and their header files; just add your .c and .h to this list.

If your driver depends on certain system headers (eg, the gzip one depends on zlib.h) then you will need to add checks for these to "configure.in" 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 data structure. Just cast the pointer to the correct type:

```
/* San-
ity check: Is this meant for our driver? */
if (self-
>cd_class != &cc_sq) return DSK_ERR_BADPTR;
sqself = (SQ_COMPRESS_DATA *)self;
```

and you're in business.

7.3 Compression functions

7.3.1 cc_open

```
dsk_err_t (*cc_open )(COMPRESS_DATA *self)
```

Attempt to decompress a compressed file.

- “self” points to the instance data for this disc image.
- self->cd_cfilename is the filename of the file to decompress.

Return:

DSK_ERR_OK: The file has been decompressed.

DSK_ERR_NOTME: The file is not compressed using this driver’s method.

other: The file does 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:

```
dsk_err_t comp_fopen(COMPRESS_DATA *self, FILE **pfp);
```

Open the the file whose name is given at `self->cd_cfilename`. If successful, `*pfp` will be the opened stream. If not, it will be NULL. If the file can only be opened read-only, sets `self->cd_readonly` to 1.

```
dsk_err_t comp_mktemp(COMPRESS_DATA *cd, FILE **pfp);
```

Create a temporary file and store its name at `self->cd_ufilename`. You should use this to create the file that you decompress into.

7.3.2 cc_creat

```
dsk_err_t (*cc_creat)(COMPRESS_DATA *cd)
```

Warn the compression engine that a disc image file is being created, and when closed it will be compressed. The filename is stored at `self->cd_cfilename`. Normally this just returns DSK_ERR_OK.

7.3.3 cc_commit

```
dsk_err_t (*cc_commit)(COMPRESS_DATA *cd)
```

Compress an uncompressed file. `self->cd_ufilename` is the name of the file to compress. `self->cd_cfilename` is the name of the output file.

7.3.4 cc_abort

```
dsk_err_t (*cc_abort)(COMPRESS_DATA *cd)
```

This is used if a file was decompressed and it’s now being closed without having been changed. There is therefore no need to compress it again. This normally just returns DSK_ERR_OK.

A DQK Files

A DQK file is a .DSK file compressed using Richard Greenlaw's Squeeze file format (originally from CP/M as SQ.COM, and later built in to NSWP.COM; 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 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 distribution. It contains the following files:

- dskwrite.com: Program to write .DSK or .DQK files out to a real 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 or LGPLed. They are public domain. You may do whatsoever you please with them.

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

B LibDsk with cpmtools

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

The myz80 and nanowasp drivers use a fixed disk format; here are diskdefs entries which can be used to read them (these are for the old diskdefs format with

one line per entry):

```
myz80    1024 64 128 4096 1024 1 0 3
microbee  512 80  10 2048 128  1 2 2.2
```