|  |  |  |
| --- | --- | --- |
| Bochs Developers Guide | | |
| [Prev](http://docs.google.com/sound-lowlovel-basics.html) | Chapter 2. About the code | [Next](http://docs.google.com/add-keymapping.html) |

2.10. Harddisk Images based on redologs

This section describes how the three new disk images "undoable", "growing", and "volatile" are implemented in Bochs 2.1. It also applies to the write support the "vvfat" disk image mode in Bochs 2.4.6.

* undoable -> base r/o file, plus growing, commitable, rollbackable redolog file
* growing -> growing files, all previously unwritten sectors go to the end of file
* volatile -> base r/o file, plus hidden growing redolog
* vvfat -> virtual VFAT disk created from directory, plus hidden growing redolog

2.10.1. Description

The idea behind volatile and undoable disk images is to have a read-only base file, associated with one redolog file. In case of vvfat, a directory is associated with the redolog file.

Reading a sector is done from the redolog file if it contains the sector, or from the base file / vvfat directory otherwise.

Sectors written go to the redolog, so base image files are opened in read only mode in this configuration.

The redolog is designed in a way so it starts as a small file and grows with every new sectors written to it. Previously written sectors are done in place. Redolog files can not shrink.

The redolog is a growing file that can be created on the fly.

Now, it turns out that if you only use a redolog without any base image file, you get a "growing" disk image.

So "undoable", "volatile", "growing" and "vvfat" harddisk images classes are implemented on top of a redolog class.

2.10.2. How redologs works ?

At the start of a redolog file, there is a header, so Bochs can check whether a file is consistent. This header is also checked when the automatic type and size detection is selected.

The generic part of the header contains values like type of image, and spec version number.

The header also has a specific part. For redologs, the number of entries of the catalog, the extent, bitmap and disk size are stored.

In a redolog, the disk image is divided in a number of equal size "extents". Each extent is a collection of successive 512-bytes sectors of the disk image, preceeded by a n\*512bytes bitmap.

the n\*512bytes bitmap defines the presence (data has been written to it) of a specific sector in the extent, one bit for each sector. Therefore with a 512bytes bitmap, each extent can hold up to 4k blocks

Typically the catalog can have 256k entries. With a 256k entries catalog and 512bytes bitmaps, the redolog can hold up to 512GiB

**Note:** All data is stored on images as little-endian values

2.10.2.1. Header

At the start of a redolog file, there is a header. This header is designed to be reusable by other disk image types.

The header length is 512 bytes. It contains :

**Table 2-8. Generic header description**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Start position in bytes | Length in bytes | Data type | Description | Possible values |
| 0 | 32 | string | magical value | Bochs Virtual HD Image |
| 32 | 16 | string | type of file | Redolog |
| 48 | 16 | string | subtype of file | Undoable, Volatile, Growing |
| 64 | 4 | Bit32u | version of used specification | 0x00010000, 0x00020000 |
| 68 | 4 | Bit32u | header size | 512 |

The current version of the header is 0x00020000 (2.0) - see below for details.

**Table 2-9. Redolog specific header description**

|  |  |  |  |
| --- | --- | --- | --- |
| Start position in bytes | Length in bytes | Data type | Description |
| 72 | 4 | Bit32u | number of entries in the catalog |
| 76 | 4 | Bit32u | bitmap size in bytes |
| 80 | 4 | Bit32u | extent size in bytes |
| 84 | 4 | Bit32u | timestamp in FAT format ("undoable" mode only - otherwise reserved) |
| 88 | 8 | Bit64u | disk size in bytes |

The reserved field between "extent" and "disk" has been added in redolog version 2.0 to fix an alignment bug on some platforms. It is now used for consistency check of the "undoable" mode. When creating the redolog file, the timestamp of the read-only file is stored there (in FAT format). After that, the "undoable" mode init code compares the timestamp of the r/o file with the one stored in the redolog.

2.10.2.2. Catalog

Immediately following the header, there is a catalog containing the position number (in extents) where each extent is located in the file.

Each position is a Bit32u entity.

2.10.2.3. Bitmap

Each extent starts with a bitmap block of n\*512 bytes size. Each byte of the bitmap stores the write status of 8 coresponding disk sectors in the extent (1 = data written).

2.10.2.4. Extent

This is a collection of successive 512-bytes sectors of the disk image. The bitmap preceeding this data block contains the write status of each sector.

2.10.3. Parameters

The following tables shows what parameters are used when creating redologs or creating "growing" images :

**Table 2-10. How number of entries in the catalog and number of blocks by extents are computed**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Catalog entries | Catalog size(KiB) | Bitmap size (B) | Extent size (KiB) | Disk Max Size |
| 512 | 2 | 1 | 4 | 2MiB |
| 512 | 2 | 2 | 8 | 4MiB |
| 1k | 4 | 2 | 8 | 8MiB |
| 1k | 4 | 4 | 16 | 16MiB |
| 2k | 8 | 4 | 16 | 32MiB |
| 2k | 8 | 8 | 32 | 64MiB |
| 4k | 16 | 8 | 32 | 128MiB |
| 4k | 16 | 16 | 64 | 256MiB |
| 8k | 32 | 16 | 64 | 512MiB |
| 8k | 32 | 32 | 128 | 1GiB |
| 16k | 64 | 32 | 128 | 2GiB |
| 16k | 64 | 64 | 256 | 4GiB |
| 32k | 128 | 64 | 256 | 8GiB |
| 32k | 128 | 128 | 512 | 16GiB |
| 64k | 256 | 128 | 512 | 32GiB |
| 64k | 256 | 256 | 1024 | 64GiB |
| 128k | 512 | 256 | 1024 | 128GiB |
| 128k | 512 | 512 | 2048 | 256GiB |
| 256k | 1024 | 512 | 2048 | 512GiB |
| 256k | 1024 | 1024 | 4096 | 1TiB |
| 512k | 2048 | 1024 | 4096 | 2TiB |
| 512k | 2048 | 2048 | 8192 | 4TiB |
| 1024k | 4096 | 2048 | 8192 | 8TiB |
| 1024k | 4096 | 4096 | 16384 | 16TiB |
| 2048k | 8192 | 4096 | 16384 | 32TiB |

2.10.4. Redolog class description

The class *redolog\_t();* implements the necessary methods to create, open, close, read and write data to a redolog. It also contains methods for the subtype and consistency check and for the save/restore support. Managment of header catalog and sector bitmaps is done internally by the class.

2.10.4.1. Constants

#define STANDARD\_HEADER\_MAGIC "Bochs Virtual HD Image"  
#define STANDARD\_HEADER\_VERSION (0x00020000)  
#define STANDARD\_HEADER\_SIZE (512)

These constants are used in the generic part of the header.

#define REDOLOG\_TYPE "Redolog"  
#define REDOLOG\_SUBTYPE\_UNDOABLE "Undoable"  
#define REDOLOG\_SUBTYPE\_VOLATILE "Volatile"  
#define REDOLOG\_SUBTYPE\_GROWING "Growing"

These constants are used in the specific part of the header.

#define REDOLOG\_PAGE\_NOT\_ALLOCATED (0xffffffff)

This constant is used in the catalog for an unwritten extent.

2.10.4.2. Methods

*redolog\_t();* instanciates a new redolog.

*int make\_header(const char\* type, Bit64u size);* creates a header structure in memory, and sets its *type* and parameters based on the disk image *size*. Returns 0.

*int create(const char\* filename, const char\* type, Bit64u size);* creates a new empty redolog file, with header and catalog, named *filename* of type *type* for a *size* bytes image. Returns 0 for OK or -1 if a problem occured.

*int create(int filedes, const char\* type, Bit64u size);* creates a new empty redolog file, with header and catalog, in a previously opened file described by *filedes*, of type *type* for a *size* bytes image. Returns 0 for OK or -1 if a problem occured.

*int open(const char\* filename, const char\* type, Bit64u size);* opens a redolog file named *filename*, and checks for consistency of header values against a *type* and *size*. Returns 0 for OK or -1 if a problem occured.

*int open(const char\* filename, const char\* type, Bit64u size, int flags);* opens a redolog file with *flags* applied. This allows to open a redolog in read-only mode. All other parameters and the return value are similar to the default *open()* method above.

*void close();* closes a redolog file.

*off\_t lseek(off\_t offset, int whence);* seeks at logical data offset *offset* in a redolog. *offset* must be a multiple of 512. Only SEEK\_SET and SEEK\_CUR are supported for *whence*. Returns -1 if a problem occured, or the current logical offset in the redolog.

*ssize\_t read(void\* buf, size\_t count);* reads *count* bytes of data of the redolog, from current logical offset, and copies it into *buf*. *count* must be 512. Returns the number of bytes read, that can be 0 if the data has not previously be written to the redolog.

*ssize\_t write(const void\* buf, size\_t count);* writes *count* bytes of data from *buf* to the redolog, at current logical offset. *count* must be 512. Returns the number of bytes written.

*Bit64u get\_size();* returns the size stored in the "disk" field in the header. This is used for size autodetection feature ("growing" mode) and the consistency check ("undoable" mode).

*Bit32u get\_timestamp();* returns the value of the "timestamp" field in the header (only used by the "undoable" mode).

*bx\_bool set\_timestamp(Bit32u timestamp);* writes the *timestamp* to the header. This is done by the "undoable" mode init code if *get\_timestamp()* returns 0 or the redolog is newly created.

*static int check\_format(int fd, const char \*subtype);* checks the format of the file with descriptor *fd*. Returns *HDIMAGE\_FORMAT\_OK* if the *subtype* matches the requested one. This is used for for the image mode autodetection feature.

*bx\_bool save\_state(const char \*backup\_fname);* copies the redolog file to a new file *backup\_fname*. This is used by the hdimage save/restore feature.

2.10.5. Disk image classes description

"volatile" and "undoable" disk images are easily implemented by instanciating a *device\_image\_t* object (base image) and a *redolog\_t* object (redolog).

"growing" disk images only instanciates a *redolog\_t* object.

Class names are *undoable\_image\_t*, *volatile\_image\_t* and *growing\_image\_t*.

When using these disk images, the underlying data structure and layout is completely hidden to the caller. Then, all offset and size values are "logical" values, as if the disk was a flat file.

2.10.5.1. Constants

#define UNDOABLE\_REDOLOG\_EXTENSION ".redolog"  
#define UNDOABLE\_REDOLOG\_EXTENSION\_LENGTH (strlen(UNDOABLE\_REDOLOG\_EXTENSION))  
#define VOLATILE\_REDOLOG\_EXTENSION ".XXXXXX"  
#define VOLATILE\_REDOLOG\_EXTENSION\_LENGTH (strlen(VOLATILE\_REDOLOG\_EXTENSION))

These constants are used when building redolog file names

2.10.5.2. undoable\_image\_t methods

*undoable\_image\_t(Bit64u size, const char\* redolog\_name);* instanciates a new *undoable\_image\_t* object. This disk image logical length is *size* bytes and the redolog filename is *redolog\_name*.

*int open(const char\* pathname);* opens the disk image *pathname* in read-only mode, as an undoable disk image. The image mode of this base image is auto-detected. All supported disk image modes can be used here. The associated redolog will be named *pathname* with a *UNDOABLE\_REDOLOG\_EXTENSION* suffix, unless set in the constructor. Returns 0 for OK or -1 if a problem occured.

*void close();* closes the base image and its redolog.

*off\_t lseek(off\_t offset, int whence);* seeks at logical data position *offset* in the undoable disk image. Only SEEK\_SET and SEEK\_CUR are supported for *whence*. Returns -1 if a problem occured, or the current logical offset in the undoable disk image.

*ssize\_t read(void\* buf, size\_t count);* reads *count* bytes of data from the undoable disk image, from current logical offset, and copies it into *buf*. *count* must be 512. Returns the number of bytes read. Data will be read from the redolog if it has been previously written or from the base image otherwise.

*ssize\_t write(const void\* buf, size\_t count);* writes *count* bytes of data from *buf* to the undoable disk image, at current logical offset. *count* must be 512. Returns the number of bytes written. Data will always be written to the redolog.

*bx\_bool save\_state(const char \*backup\_fname);* calls the related redolog\_t method to save the image state.

*void restore\_state(const char \*backup\_fname); called by the hdimage restore code. Copies the backup file to the original location and overwrites the existing redolog file.*

2.10.5.3. volatile\_image\_t methods

*volatile\_image\_t(Bit64u size, const char\* redolog\_name);* instanciates a new *volatile\_image\_t* object. This disk image logical length is *size* bytes and the redolog filename is *redolog\_name* plus a random suffix.

*int open(const char\* pathname);* opens the disk image *pathname* in read-only mode, as a volatile disk image. The image mode is auto-detected. The associated redolog will be named *pathname* with a random suffix, unless set in the constructor. Returns 0 for OK or -1 if a problem occured.

*void close();* closes the base image and its redolog. The redolog is deleted/lost after close is called.

*off\_t lseek(off\_t offset, int whence);* seeks at logical data position *offset* in the volatile disk image. Only SEEK\_SET and SEEK\_CUR are supported for *whence*. Returns -1 if a problem occured, or the current logical offset in the volatile disk image.

*ssize\_t read(void\* buf, size\_t count);* reads *count* bytes of data from the volatile disk image, from current logical offset, and copies it into *buf*. *count* must be 512. Returns the number of bytes read. Data will be read from the redolog if it has been previously written or from the base image otherwise.

*ssize\_t write(const void\* buf, size\_t count);* writes *count* bytes of data from *buf* to the volatile disk image, at current logical offset. *count* must be 512. Returns the number of bytes written. Data will always be written to the redolog.

*bx\_bool save\_state(const char \*backup\_fname);* calls the related redolog\_t method to save the image state.

*void restore\_state(const char \*backup\_fname); called by the hdimage restore code. Copies the backup file to the original location and overwrites the existing redolog file.*

2.10.5.4. growing\_image\_t methods

*growing\_image\_t(Bit64u size);* instanciates a new *growing\_image\_t* object. This disk image logical length is *size* bytes.

*int open(const char\* pathname);* opens the growing disk image *pathname*, Returns 0 for OK or -1 if a problem occured.

*void close();* closes the growing disk image.

*off\_t lseek(off\_t offset, int whence);* seeks at logical data position *offset* in the growable disk image. Only SEEK\_SET and SEEK\_CUR are supported for *whence*. Returns -1 if a problem occured, or the current logical offset in the grwoing image.

*ssize\_t read(void\* buf, size\_t count);* reads *count* bytes of data from the growing disk image, from current logical offset, and copies it into *buf*. *count* must be 512. Returns the number of bytes read. The buffer will be filled with null bytes if data has not been previously written to the growing image.

*ssize\_t write(const void\* buf, size\_t count);* writes *count* bytes of data from *buf* to the growing disk image, at current logical offset. *count* must be 512. Returns the number of bytes written.

*static int check\_format(int fd, Bit64u imgsize);* checks the format of the file with descriptor *fd*. Returns *HDIMAGE\_FORMAT\_OK* if the file format matches the "growing" one. This is used for the image mode autodetection feature.

*bx\_bool save\_state(const char \*backup\_fname);* calls the related redolog\_t method to save the image state.

*void restore\_state(const char \*backup\_fname); called by the hdimage restore code. Copies the backup file to the original location and overwrites the existing redolog file.*

|  |  |  |
| --- | --- | --- |
| [Prev](http://docs.google.com/sound-lowlovel-basics.html) | [Home](http://docs.google.com/index.html) | [Next](http://docs.google.com/add-keymapping.html) |
| The sound lowlevel interface | [Up](http://docs.google.com/about-the-code.html) | How to add keymapping in a GUI client |