Bit Recovery Documentation

Release 1.2.2

Dirk Roorda

September 23, 2015

Contents

1 ABOUT **3**
- 1.1 Data in various stages of decay and salvation . . . 3
- 1.2 Account . . . 7
- 1.3 Contents . . . 7

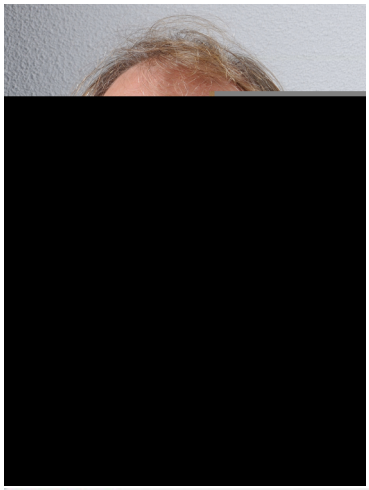
2 checksum.pl **9**
- 2.1 Description . . . 9
- 2.2 Usage . . . 9
- 2.3 Generating . . . 10
- 2.4 Verifying . . . 10
- 2.5 Repairing . . . 10
- 2.6 Restoring . . . 11
- 2.7 Executing . . . 11
- 2.8 Diagnostics . . . 11
- 2.9 Author . . . 11
- 2.10 Configuration . . . 12
- 2.11 Implementation details . . . 12

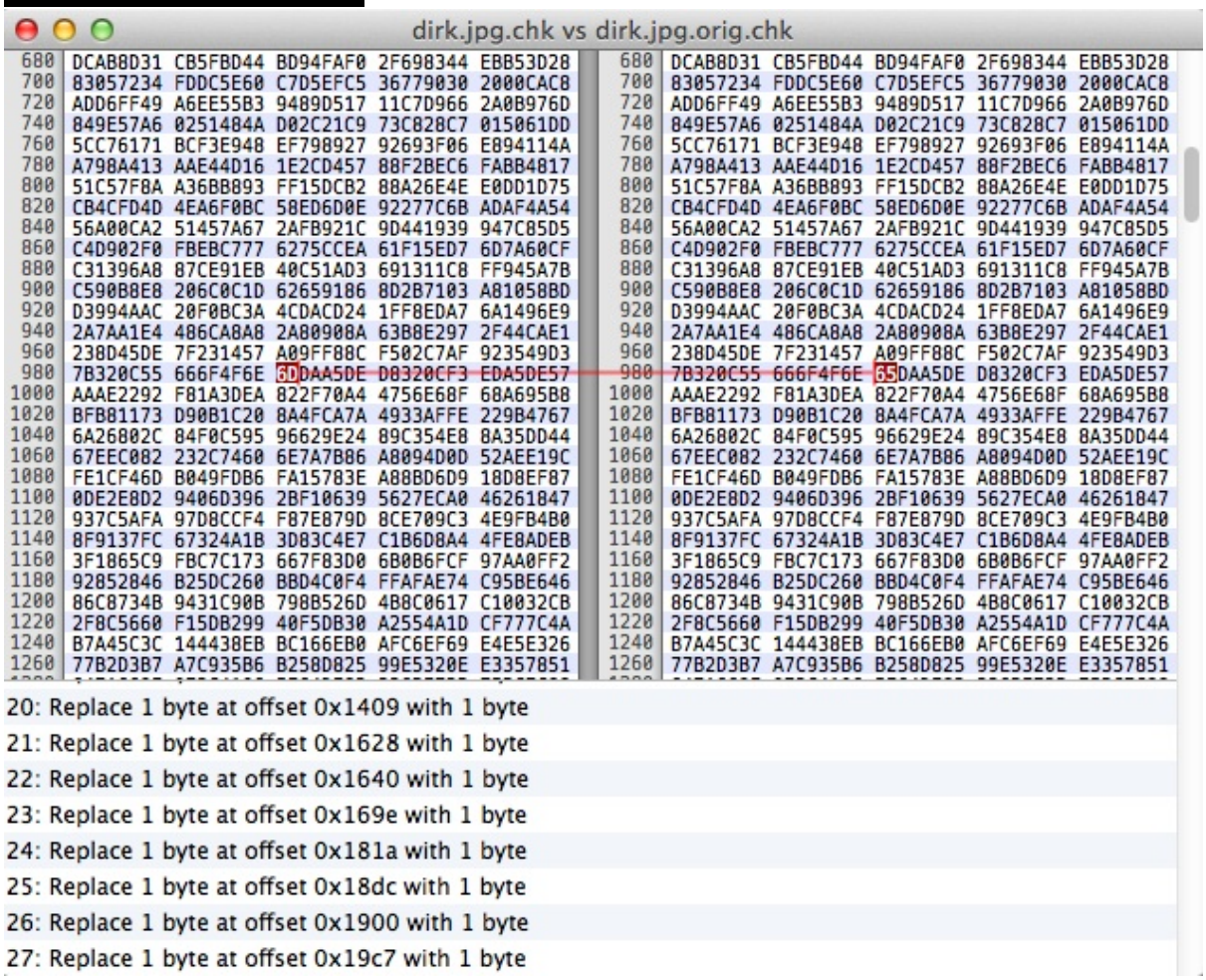
3 perfset.pl **15**
- 3.1 Description . . . 15
- 3.2 Usage . . . 15

4 corrupt.pl **17**
- 4.1 Description . . . 17
- 4.2 Usage . . . 17

5 gather.pl **19**
- 5.1 Description . . . 19
- 5.2 Usage . . . 19

6 Indices and tables **21**




dirk.jpg.chk vs dirk.jpg.orig.chk
20: Replace 1 byte at offset 0x1409 with 1 byte
21: Replace 1 byte at offset 0x1628 with 1 byte
22: Replace 1 byte at offset 0x1640 with 1 byte
23: Replace 1 byte at offset 0x169e with 1 byte
24: Replace 1 byte at offset 0x181a with 1 byte
25: Replace 1 byte at offset 0x18dc with 1 byte
26: Replace 1 byte at offset 0x1900 with 1 byte
27: Replace 1 byte at offset 0x19c7 with 1 byte

Contents:

CHAPTER 1

ABOUT

1.1 Data in various stages of decay and salvation

When you store TeraBytes of data for many years, some bits in it will decay. It is hard to get figures about how much damage we can expect. But it might be in the order of a handful per TB per year.

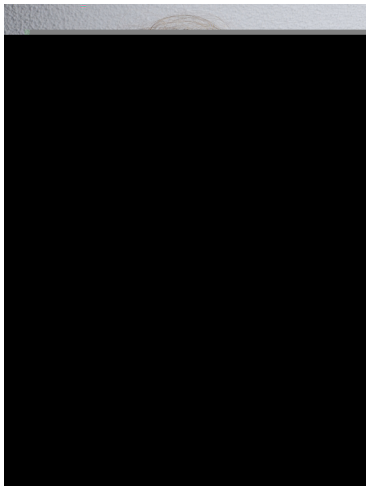
How do we recover from it? Here is an elegant method. Add some redundancy, in the form of checksums. Periodically check the checksums. When there are errors, use the checksums to correct the errors, if possible. If it is not possible, use a backup. But beware: the backup might have errors as well. Even the checksums themselves might have errors. Before we explain our strategy, here is an example that it actually works.

1. original

We start with a photo of the author. It is a 436 KB jpeg image. This is indeed the uncorrupted form.

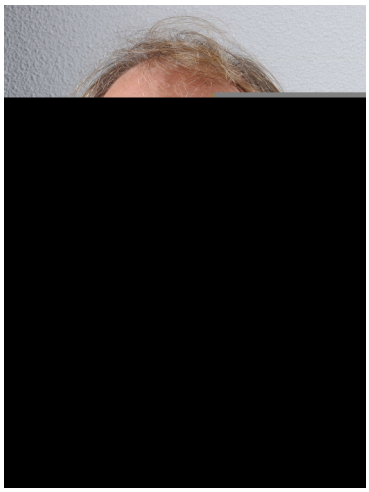
2. 174 bit errors

Now 174 bit errors are added, at random positions



3. 104 errors in the backup

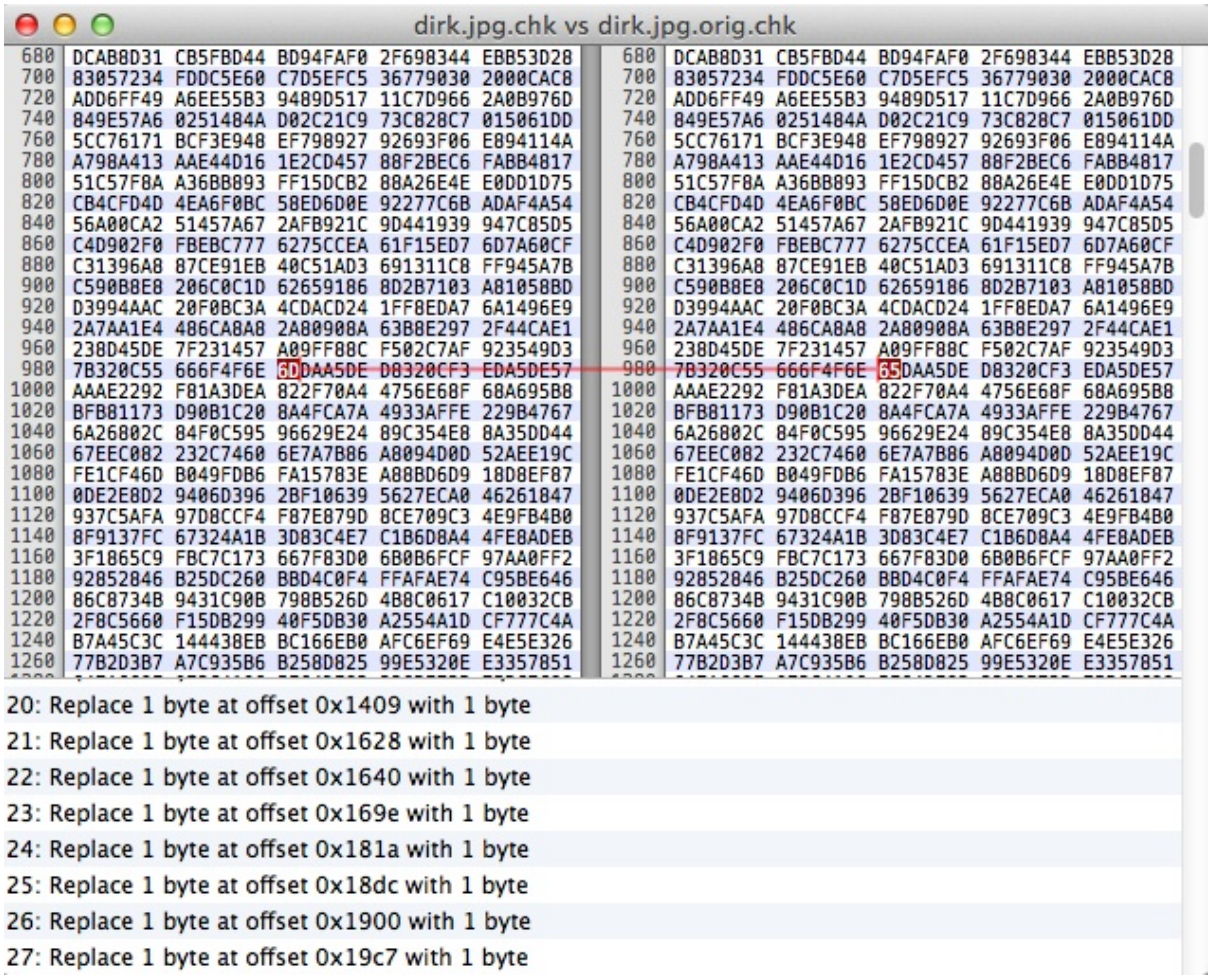
We will also use a backup, but also this one is corrupted: 104 different errors



4-5. 27 + 16 bit errors in the checksum files

We also corrupt the checksums: 27 bit errors

```
Info: /Users/dirk/Scratch/dirk.jpg.chk has 55936 bits
Need to generate 27 bit errors for [/Users/dirk/Scratch/dirk.jpg.chk]
```



and the checksums of the backup are not spared either: 16 bit errors

```
Info: /Users/dirk/Scratch/dirk-bu.jpg.chk has 55936 bits
Need to generate 16 bit errors for [/Users/dirk/Scratch/dirk-bu.jpg.chk]
```

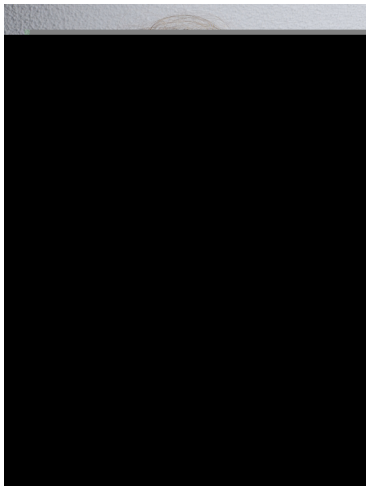
6. checking the corrupt image with the corrupt checksums

we get 163 damaged blocks

```
Info: Verification results:
        Total blocks: 436
        Good blocks: 273
        Damaged blocks: 163
```

7. after repairing 138 and leaving 25 bit errors

First we try to repair without using the backup, we can repair the majority of damaged blocks, 138.



But 25 remain unrepaired. See the result.

```
Info: Repair results:
        Repaired blocks: 138
        Unrepaired blocks: 25
        Suspicion level: 0
```

Let us again check the checksums. 50 damaged blocks! But remember that the checksums themselves were faulty!

```
Info: Verification results:
        Total blocks: 436
        Good blocks: 386
        Damaged blocks: 50
```

Yet, by a combination of restoring and repairing it is effectively possible to correct all errors.

8. We need to use the backup

```
Info: Restore results:
        Restored blocks: 50
        Unrestored blocks: 0
        Suspicion level: 0
```

9. fully restored, thank you

10. There are absolutely no errors left

```
dirk:~/Scratch > diff -s dirk-orig.jpg dirk-restored.jpg
Files dirk-orig.jpg and dirk-restored.jpg are identical
```

1.2 Account

This story I wrote on Good Friday, 2013-03-29. All data and screenshots were directly taken from the computer when I executed the process as described above. By the way, that was a Macbook Air, and the whole process is expressed in a Perl script, which only uses the module Digest:MD5. Both Perl and this module are already present in OSX.

1.3 Contents

After that, I have tested extensively. The code for this lab is in Github.

The report I published on Figshare.

It is a tool for checksumming files in such a way that you can recover from errors. It also does the recovering. Besides, it is an environment to test various checksumming algorithms and parameters to see what performs best. You find also test data of a few dozens of experiments, summarized in an excel document. The code is here (Perl).

There is a program for checksumming files, verifying, repairing and restoring: *checksum.pl*.

Then there is a setup to do experiments: *perfset.pl* creates a pool of corrupt file and organizes tests of various checksum methods.

The question is: wich checksum methods *perform* best in the brute force search for the original byte sequence?

In order to make file corrupt, you can run *corrupt.pl* with a variety of parameters.

To gather the results of a series of experiments, use *gather.pl*. It creates a csv file, that you can use to create nice graphics in a spreadsheet program.

CHAPTER 2

checksum.pl

2.1 Description

This tool is an instrument in bit preservation of (large) files. It is estimated that if one reads 10 TB from disk, 1 bit will be in error. Also, when 1 TB is stored for a year without touching it, some bits might be damaged by random physical events such as radiation.

In order to bit-peserve large files for longer periods of time (years, decades), it becomes important to guard against data loss.

While there is no profound solution to this problem, the following stratgegy counts as best practice.

- Make several copies
- Divide the file and their copies in chuncks and compute checksums of the chunks
- periodically check checksums and restore damaged blocks from copies where the corresponding block is undamaged.

Checksum.pl is a script to compute checksums for files, to verify checksums, and to repair corrupted file by means of brute force searching, or if that is not feasible, by restoring from backup copies, even if those are corrupt themselves. It works even when the checksums themselves are corrupt.

It al depends on the damage being not too big.

2.2 Usage

Call the script like this:

```
./checksum.pl [-v] [-m method] [-t task]* [--conf kind=path]* --data kind=path [backupfile] [orig
```

where

```
-v                      verbose operation
method          key of %config_checksum
task            member of:
                                generate
                                verify
                                repair
                                restore
                                restore_ambi_no
                                restore_ambi_only
                                execute_repair
                                execute_restore
                                diag

conf:
```

```
        kind    key of %files
        path    will replace the name value in %files

data
        kind    key of %datafile
        path    path to a file on the file system
```

This script can generate checksums, verify them, and perform repair and restore from backup. The verification step produces a file with mismatches, if present. The repair and restore steps look at the file with mismatches and then try to find out how to repair those mismatches. The result is writen to a file with instructions. An execute step reads those instructions and executes them, actually changing the data file. The checksum files are not modified. They can easily be recomputed again.

All intermediate files (also those with the generated checksums) are binary: all data consists of fixed length strings, 64-bit integers, or fixed-size blocks of binary data. All these files have a header, indicating the checksum method used, as well as the data block size and the checksum length.

With arguments like *file:kind=path* you can overrule the locations and names (but not extensions) of all files that are read and written to. The kind part must occur as key in the *%files* hash.

2.3 Generating

Command:

```
./checksum.pl file
```

Generates checksums for (large) files, block by block. The size of a block is configured to 1_000 bytes. The main reason to keep it fairly small is to be able to do brute force guessing when a checksum is found not to agree anymore with a datablock.

By generating many slight bit errors in the datablock as well as the checksum, and then searching for a valid combination of datablock and checksum, we can be nearly completely sure that we have the original datablock and checksum back.

The file with checksums has the same name as the input file, but with *.chk* appended to it.

2.4 Verifying

Command:

```
./checksum.pl -v file
```

Verifies given checksums. It expects next to the input file a *file.chk* with checksums, in the format indicated above. It then extracts from file each block as specified in *file.chk*, computes its checksum and compares it to the given checksum.

If there are checksum errors, references to the blocks in error are written to an error file, with name *file.x* . This file contains records of mismatch information. Such a record consists of just the block number, the given checksum, and the computed checksum.

If there are no errors, the *file.x* will not be present. If it existed, it will be deleted.

2.5 Repairing

Command:

```
./checksum.pl -c file
```

Looks at checksum mismatches. In every case, modifies checksum and corresponding blocks in many small ways, until the combination matches again. Both block and checksum are dithered. That means, a frame of at most n bits wide moves over the data, and inside the frame the bits are mangled in all possible ways. The dither results of the checksum are stored in a hash. The dithered blocks are not stored. They are generated on the fly, their checksum is computed, and quickly tested against the hash of checksums. If there is a hit, it will be stored. If there are no hits, repair is not possible by the current method. You might try further by increasing the frame width, or by trying other kinds of variants of the block. But maybe it is better to forget this method and try to restore from backup in such cases. If there are multiple hits, that would be a weird situation. Maybe there has been intentional tampering. The program will give clear warnings in these cases.

The repair instructions are written to *file.ri*

2.6 Restoring

Command:

```
./checksum.pl -r[a|A] file file-backup
```

Compares blocks and checksums of data and backup. The bit positions where they differ, will be varied among all possibilities. The checksums are stored in a hash for easy lookup. Then the blocks will be generated on the fly. So even if the backup is damaged, and even if the checksums are all damaged, it is still possible by brute force search to find the original data back. If data and backup differ in less than 20 bits per block, there are only a million possibilities per block to be searched. If called with -rA only the blocks for which repair found multiple hits will be restored (not the ones without hits) If called with -ra both the blocks for which repair found multiple hits and no hits will be restored

The restore instructions are written *file.rib*

2.7 Executing

Commands:

```
./checksum.pl -ec file
./checksum.pl -er file
```

Executes the repair resp. restore instructions in *file.ri* resp. *file.rib* All information needed from the backup file is already in the instruction file, so the backup file itself is not needed here. The work has been done in the previous steps, this step only performs the write actions in the file.

2.8 Diagnostics

Command:

```
./checksum.pl -dia file backupfile origfile corruptfile
```

Creates a diagnostic report of the repair and restore instructions. It takes as second argument the backup file and as third argument the original file and as fourth argument the unrestored/unrepaired corrupted file. It gives all info about the blocks which have not been restored correctly. On the basis of this information it shows which instructions helped to correctly get the original back, and which instructions were faulty.

2.9 Author

Dirk Roorda, Data Archiving and Networked Services (DANS) 2013-03-29 dirk.roorda@dans.knaw.nl

See also DANS Lab Bit rot and recovery

2.10 Configuration

In order to compare performance between md5 and sha256 hashing we provide two standard configurations, which can be invoked by the command line flag `-m`:

```
-m md5
-m sha256
```

invoke the md5 and the sha256 checksum algorithms respectively. The default parameter values for these methods are loaded. It remains possible to overrule these values by means of additional flags on the command line.

The default checksum mode is sha256.

2.11 Implementation details

2.11.1 Looking for hits

When measuring how close a "hit" is to the actual situation, the number of different bits in the checksums and in the blocks are counted. However, differences in the checksum count much more than differences in the blocks.

Bit differences in the checksums are far less probable than bit differences in the blocks, because blocks are larger. Moreover, if checksums are very different, it is an indication of tampering: a new checksum has been computed for a slightly altered block. So by default we multiply the checksum bit distance by the $data_checksum_ration. In addition, you can configure to increase or decrease this effect by multiplying with the $check_diff_penalty which is by default 1.

We compare hits with the foreground file, not with the backup. We want a hit that is closest to the foreground, since the foreground has been always under our control, and the backup has been far less in our control.

We want to keep the search effort constant for the different checksum methods. Depending on the blocksize determined by the checksum method, we can set the search parameters in such a way that the prescribed number of search operations will be used.

2.11.2 Binary files and headers

Every binary non-data file we read, is a file generated by this program. Such a file has a header. It will be read and written by the following two functions. It has the format:

```
a8 a8 L L L L
```

where:

```
a8 is arbitrary binary data of 8 bytes. Reserved for a string indicating the checksum method
a8 is arbitrary binary data of 8 bytes. Reserved for a string indicating the checksum method
L is a long integer (32 bits = 4 bytes), indicating the checksum size
L is a long integer (32 bits = 4 bytes), indicating the checksum size
L is a long integer (32 bits = 4 bytes), indicating the block size
L is a long integer (32 bits = 4 bytes), indicating the block size
```

All together the header is 32 bytes = 256 bits

The header could be damaged. We assume the checksum size and the block size are powers of two. If one of them does not appear a power of two, choose the other. If both are not powers of two, we are stuck. If both are powers of two but different, we are also stuck. Likewise, we choose between the values encountered for the checksummethod.

2.11.3 Reading and Writing files

Opens files for reading, writing, and read-writing. Uses the specification created in the init() function. Returns a file handle in case of succes. The file handle is meant to be stored in global variables. So more than one routine can easily read and write the same file.

2.11.4 Repair block

This function implements a main step: Repair a single block We apply ditherings progressively, in rounds corresponding to the frame length n of the dithering. We start with n = 0, then n = 1 and so on. So the smaller disturbances will be checked first, and we assume that bigger disturbances do not compete with smaller ones. If there are hits in a round, the next rounds will be skipped.

2.11.5 Restore block

now generate the set by creating all possible bit values at the positions where $str1 and $str2 differ in order to optimize the search process, we want to search in such a way that we do cases first where bits are taken consecutively from the data version or the backup version. The reason is that errors come in bursts. Hence, if backup and data differ in bit i and bit i+1, both bits are likely to be correct in either backup or in data. It is much less likely that bit i is correct in data and bit i+1 in backup, or vice versa. So if the max number of brute force operations does not permit full traversal, we do a partial traversal with the most likely suspects first. This will increase the change of finding a good restore.

So we generate all possibile bit strings for the difference mask. We will xor the bits in the mask with the corresponding bits in the data. So we should try bitstrings first with minimal alterations between 1s and 0s.

2.11.6 Dithering

This is the technique used for repairing blocks.

Dithering is subtly mangling a bit string, by introducing a limitied amount of bit errors. We let an imaginary frame of fixed width slide over the bitstring, and inside the frame we generate all possible bit errors.

More precisely, n-dithering is dithering with a frame of exactly width n. And <=n-dithering is dithering with frames of width 1 to n.

If we do n-dithering, we generate bitstrings of length n, and x-or the input bitstring with it, at a reference position that slides throughout the input.

Bit 0 and bit n-1 of an n-frame are always 1. If one or of them would be 0, we would have an n-1 frame, or an n-2 frame, or even less. We would be doing double work then.

Bits 1 up to and including n-2 range over the full set of possible bitstrings of length n-2.

n-ditherings and m ditherings are mutually exclusive when n <> m. This is precisely because the end points are always one, and the endpoints change the input bitstring.

So the number of ditherings with frame length <= n is: 2 ^ (n-1)

2.11.7 Masking

This is the technique used for restoring blocks. When the corresponding block from the backup is fetched, and we have the data block, then in the most general case we do not know which block is right. They could be both wrong. Even the checksums could be all wrong.

We assume however, that the bits in which they agree are correct.

So me make a mask of the differing bits, and we create all bit variations in that mask.

We try them all out by brute force.

So there is good chance that we find a hit, even if all initial data is corrupted.

CHAPTER 3

perfset.pl

3.1 Description

Generates a test sets from a base file called dataname-orig in a root directory. The root directory and some other parameters are defined by the experiment. There are several experiments spelled out below, the first argument selects a specific one. An original data file is corrupted and copied to form the starting point of several parts of the test set. Each part correspondes to a checksum method such as md5 or sha256. Corruption is pseudo random, no two corruptions will be the same. From then on both parts will be subjected to checksum tests and error correcting.

3.2 Usage

Command:

```
./perfset.sh [-v] [-v] [-d] -e experiment [-tm timestamp]
```

where

```
-v                              verbose rsync, if twice: verbose all
-d                              debug mode when calling perl scripts
-f                              force fresh corruption
-c                              execute the changes and perform final check
-e experiment   key of %experiment
```

CHAPTER 4

corrupt.pl

4.1 Description

Corrupts the file with (burst)bit errors. If level is given, it is the desired number of (burst)bit errors per TB. If number is given, it is the desired absolute number of bit errors.

The bit errors are generated at independently randomly chosen positions.

It is also possible to generate burst errors of length at most nbits. A burst error is a sequence of identical bits that will overwrite a sequence of equal length in the input file. The length of the burst is determined randomly and independently but stays below the maximum length. The value of the burst (zeroes or ones) will be determined randomly.

4.2 Usage

Command:

```
./corrupt.pl [-s] [-b nbits] [-l level | -n number] --data file*
```

CHAPTER 5

gather.pl

5.1 Description

Gather data from experiments

5.2 Usage

Command:

```
./gather.pl [-v] [--base reportbasedir]
```

where

```
-v                      verbose rsync, if twice: verbose all
--base          base directory of the reports
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search