

## Parity Volume Set specification 1.0 [2001-10-14]

by Stefan Wehler - initial idea by Tobias Rieper, further suggestions and format tweaking by Kilroy Balore, Willem Monsuwe and Karl Vogel. Overview

The following document describes a file format called Parity Volume Set.

A Parity Volume Set provides RAID like parity records for a given number of files. With this parity records it is possible to restore missing files of the set.

An Example:

You have 10 data files.

data files

Foobar.d01  
Foobar.d02  
Foobar.d03  
Foobar.d04  
Foobar.d05  
Foobar.d06  
Foobar.d07  
Foobar.d08  
Foobar.d09  
Foobar.d10

With a program using this specification you can calculate parity volumes for these files.

parity volumes

Foobar.p01  
Foobar.p02  
Foobar.p03

The parity volumes contain a reed solomon checksum of the data of the data files. So if a file is missing or corrupt, it can be reconstructed out of the remaining data files and the parity volumes. Any file of this set can be reconstructed with any parity volume. You just need as much parity volumes, as files you are missing.

For instance, in the example above, Foobar.d03 is lost. So it is possible to restore Foobar.d03 using the remaining data files and one parity volume. It doesn't matter, which parity volume (p01/p02/p03) - everyone will do...

In case you miss 2 files, you need 2 parity volumes. What parity volumes and their combination

(p01+p02/p01+p03/p02+p03) doesn't matter too - you just need two of them. (They have to be different: renaming a copy of foobar.p01 to foobar.p02 and using it with foobar.p01 to restore will not work...) Description

A parity volume set consists of two parts. First there is a .PAR file (the index file).

It contains information about the files, which are "saved" in the volume sets:

- what files are stored in the set
- their checksums (MD5)
- their size
- their filename
- a general comment for the set

Second there are the parity volume files. They are named .P00, .P01, P02..... PXX. (After .P99, there will be .Q00...)

They contain:

- the same information as the PAR file (besides the comment) and
- the calculated parity data of the stored files

51000000

(The data is stored in "Intel notation".)

| Offset: | Type (Size) | Data   |
|---------|-------------|--|
| 0x0000  | 8 Bytes     | <p>Identification String<br/>"PAR"\0</p> <p>comment:<br/>-the string "PAR" followed by 5 null bytes</p>  |
| 0x0008  | 1 QWord     | <p>Version Number<br/>- The version number consists of two parts:<br/>The low Dword (bit 0..31) is the version - \$00010000 for v01.00<br/>The high Dword (bit 32..63) is a identifier for the generating program. The client can use it to see, which program generated the parity volume set. This may come in handy, if your program needs to know, if it was the generator itself. (Maybe to see, if it can use some proprietary bits in the status register or to notify, if new versions are out...)<br/>The coding is<br/>"program"-"version"-"subversion"-"subsubversion".<br/>"program" codes so far:<br/>00 - undefined<br/>01 - Mirror<br/>02 - PAR<br/>Example: Mirror 0.2.1 has \$01000201 as program identifier.</p> |
| 0x0010  | 16 Bytes    | <p>control hash<br/>-a MD5 hash of the rest of the file starting from 0x0020 to the end<br/>-used to detect a corrupt file</p>   |

|        |          |   |
|--------|----------|---|
| 0x0020 | 16 Bytes | set hash<br>-used as a identifier for the parity volume set<br>Creation:<br>Make a array of bytes (one dimention, size=used files*16).<br>Then put the MD5 hashes (not the MD516k hashes) there, starting with the first file. Use only the files, which are included in the parity data (status bit 0 = 1). Then calculate the MD5 hash of this array. |
| 0x0030 | 1 QWord  | volume number<br>-the number of the parity volume<br>-0 for the PAR file<br>-1 for .p01, 2 for .p02, 3 for .p03 etc.  |
| 0x0038 | 1 QWord  | number of files<br>-the number of files stored in the parity volume set<br>(only the input data files - the parity volumes are not counted)<br>The files, which are not stored in the parity data, but in the file list for CRC checking, are counted too. So this is the number of files in the file list.   |
| 0x0040 | 1 QWord  | start offset of the file list<br>-the start offset of the list of the stored files  |
| 0x0048 | 1 QWord  | file list size<br>-the size of the file list in bytes   |
| 0x0050 | 1 QWord  | start offset of the data<br>-the start offset of the data area  |
| 0x0058 | 1 QWord  | data size<br>-the size of the data area in bytes  |
| 0x0060 | ...      | file list<br>-This is the start of the list of the saved files.<br>-it consists of file entrys  |

A file entry has the following format:

| Offset: | Type (Size) | Data   |
|---------|-------------|--|
| 0x0000  | 1 QWord     | entry size<br>-the size of the file list entry in bytes<br>-counting starts at 1, this record is counted too |

|        |          |  |
|--------|----------|--|
| 0x0008 | 1 QWord  | status field<br>-64 bits for status flags or numbers<br>-currently only bit 0 and 1 are used:<br>bit0 = 0 - file is not saved in the parity volume set<br>bit0 = 1 - file is saved in the parity volume set<br>bit1 = 0 - file is not checked successfully yet<br>bit1 = 1 - file was successfully checked |
| 0x0010 | 1 QWord  | size<br>-the size of the stored file in bytes  |
| 0x0018 | 16 Bytes | MD5 hash<br>- a MD5 hash of the whole file   |
| 0x0028 | 16 Bytes | 16k MD5 hash<br>-a MD5 hash of the first 16384 Bytes of the file<br>-if the file is smaller then 16k, only the existing bytes are used   |
| 0x0038 | X Words  | filename<br>- the full filename with extension in Unicode ("foobar.d01")<br>- paths are not allowed<br>- no end indicators, you have to compute the size out of the entrysize (chars = (entrysize - 0x38) DIV 2)   |

After the file list, the data area starts.

For the PAR file, the data area contains a comment for the volume set. The comment is stored in uniode, control characters are allowed.

For the Pxx files, the data area contains the checksum data.

Computing of the checksum data:

You have  $n$  files and want to create  $m$  parity data. So you create a data field  $D1..Dm$ , where  $D1$  is the first byte of file1,  $D2$  is the first byte of file2,  $D3$  is the first byte of file 3...  $Dn$  is the first byte of file  $n$ . For this field you create a field of  $m$  parity checksums  $C1..Cm$ .  $C1$  is stored as the first parity byte of .p01,  $C2$  is stored as the first parity byte of .p02,  $C3$  is stored as the first parity byte of .p03...  $Cm$  is stored as the first parity byte of .pm.

Then process the second byte of all files and so on, until you reached the end of the biggest input file. If a file has no bytes left, use 0 for the calculation. So the size of the parity data will be the size of the biggest file stored in the parity volume set.

For calculating the checksums, the Reed-Solomon algorithm is used. A good description of this algorithm for programmers is <http://www.cs.utk.edu/~plank/plank/papers/CS-96-332.html>.

The "wordsize" is 8 bit, so you calculate with Bytes only. (This restricts the number of files + parity volumes to be less than 256.)

Comments:

Why the 16k MD5 hashes?

One demanded feature is, that the program will find its files even if the filenames are changed. So maybe, if the program won't find the files due to renaming, you click the "search" button and then it checks the MD5 hashes of all files in the directory to find his files. But with large numbers of large files this will take a long time. With the 16k

hashes, the program only needs to check the first 16k of each file. This will be much faster. If it finds a file, it can make a full hash too, just to be sure...

Why the file is saved - status flag?

Maybe the PAR file is only used to provide file comments and checksums. The files are not saved in a PXX volume then.

Why the file is checked - status flag?

So you can build in a function, that your program skips already successfully checked files. Repeated checking of the same set will be faster this way. (Like with QuickSFV)

---

Implementer

A file present in the file list doesn't have to be stored in the parity data at all times. So, some (or all) files can be just for checksum control or file comments in the PAR file. (In case, only a .SFV or .NFO functionality is needed...)

The saved files can be of any type and size.

There are no informations about the parity volumes stored in the header. So the program has to scan the directory for present parity volumes. They can be identified with the set hash.

There is no explicit rule for the order of the entries in the filelist, but I recommend to sort the filenames in ascending order.

---

Copyright

This document has no copyright. The introduced principles, the file format specification and its details can be used free by everyone for any type of software. There will be no license or other limitations.

It doesn't matter, if you are writing open, closed or shared source, public domain, freeware, shareware or commercial software - you can use this spec.

But if you use the PAR/PXX file format and want to make changes to the format, you have to discuss them here to maintain compatibility.

<http://www.parfiles.net>