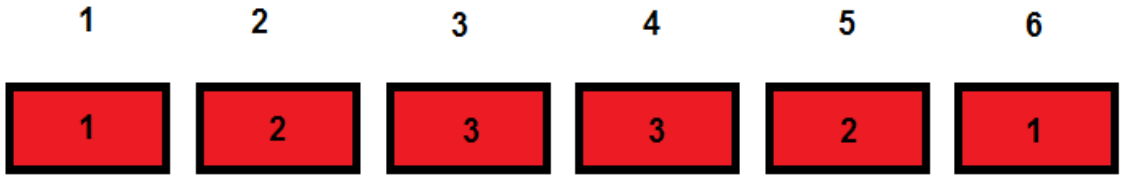
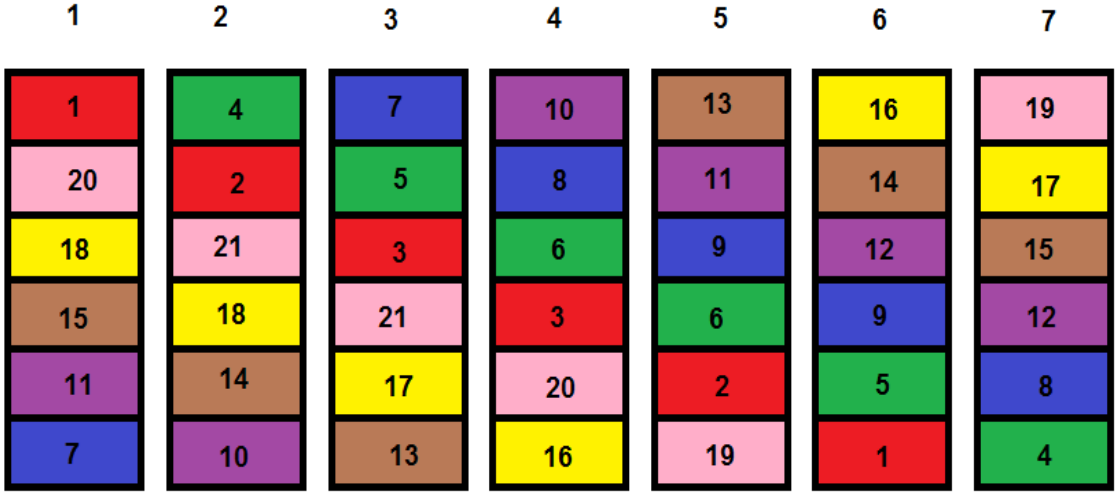
Noah Allen – Comparison of Data Recovery Designs

1. Introduction and Hypothesis

My hypothesis is that through dispersing and interleaving the data of each drive from a set of mirrored parity drive arrays into the configuration outlined in the "7 Drive Dispersed Interleaved Mirrored Parity" diagram below, it is possible to create a configuration more likely to survive 4 drives of failure (over the 6 drive configuration). In order to meet the criteria of surviving 4 drives of failure, each configuration must be able to recreate all of its data.



6 Drive Mirrored Parity



7 Drive Dispersed Interleaved Mirrored Parity

2. Method

The methods of data storage described could be developed into an implementation for physical drives, utilized in a distributed file system, or have other uses in data recovery. Because this is only the first phase in designing such a system, I will be simulating data failure for the design, but will not be testing the configuration as a potential final form.

Parity is a term in mathematics and data recovery that represents an inversion of a set of data. For the diagrams and examples used herein, I choose the highest numbered label from each data set to represent the parity. So in the "6 Drive Mirrored Parity" diagram above, the "3" label indicates the parity, which means drives 3 and 4 both contain the same parity data (the inversion of the data from labels "1" and "2"). For example, let's say I wanted to store a simple 4-byte text file, containing the text

"ABCD". The first thing required to calculate the parity would be to split the data in half; "AB" for segment 1 and "CD" for segment 2. Next an XOR (Exclusive OR or EXOR) operation would need to be performed with these 2 sets of data. The following is a breakdown of this process:

Segment 1:
"A" is 65 in ASCII decimal and 01000001 in binary.
"B" is 66 in ASCII decimal and 01000010 in binary.

Segment 2:
"C" is 67 in ASCII decimal and 01000011 in binary.
"D" is 68 in ASCII decimal and 01000100 in binary.

The XOR gate compares the 2 pieces of data and stores the result in the parity. XOR is a binary logic gate that produces a high output (1) when the sets of data are different, otherwise it will produce a low output (0).

2 Input XOR or EXOR Gate		
Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

To apply this, the first byte in segment 1 ("A" or 010000001 in binary) has an XOR operation applied against the first byte in segment 2 ("C" or 01000011 in binary).

XOR of ASCII "A" and "C"		
A	C	Output
0	0	0
1	1	0
0	0	0
0	0	0
0	0	0
0	0	0
0	1	1
1	1	0

The result for the first byte of the parity is not a printable character in ASCII (STX, start of text), 2 in decimal, and 00000010 in binary.

Next, the second byte in segment 1 (“B” or 010000010 in binary) has an XOR operation applied against the second byte in segment 2 (“D” or 01000100 in binary).

XOR of ASCII “B” and “D”		
B	D	Output
0	0	0
1	1	0
0	0	0
0	0	0
0	0	0
0	1	1
1	0	1
0	0	0

The result for the second byte of the parity is not a printable character in ASCII (ACK, acknowledge), 6 in decimal, and 00000110 in binary.

Now we have our parity data for our data set. It is as follows:

Segment 1:
“A” is 65 in ASCII decimal and 01000001 in binary.
“B” is 66 in ASCII decimal and 01000010 in binary.

Segment 2:
“C” is 67 in ASCII decimal and 01000011 in binary.
“D” is 68 in ASCII decimal and 01000100 in binary.

Segment 3:
STX is 2 in ASCII decimal and 00000010 in binary.
ACK is 6 in ASCII decimal and 00000110 in binary.

This parity data can be used to recover unknown data, in the event one of the parts of the data is missing. For example, if segment 2 is lost (leaving segment 1 and 3), we would take the first byte from segment 1 ("A" in ASCII, 65 in decimal) and perform an XOR operation with the first byte of the parity segment 3 (STX in ASCII, 2 in decimal) to recover the first byte from segment 2.

XOR of ASCII "A" and STX		
A(65)	STX(2)	Output
0	0	0
1	0	1
0	0	0
0	0	0
0	0	0
0	0	0
0	1	1
1	0	1

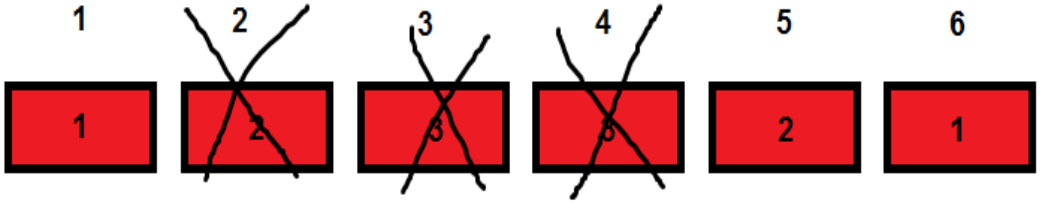
The result is the first byte of the missing segment 2 ("C" in ASCII, 3 in decimal, and 01000011 in binary). The same method is used to obtain the second missing byte from segment 2:

XOR of ASCII "B" and ACK(6)		
B	ACK(6)	Output
0	0	0
1	0	1
0	0	0
0	0	0
0	0	0
0	1	1
1	1	0
0	0	0

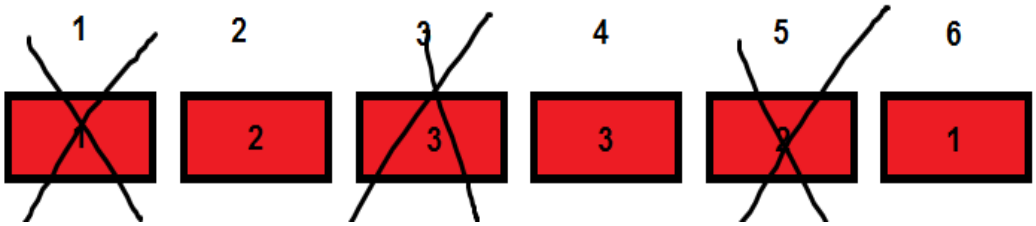
The result is the second byte of the missing segment 2 ("D" in ASCII, 4 in decimal, and 01000100 in binary). Now we have recovered segment 2.

This process will work independently of which segment is lost. For 3 pieces of data, any one piece can be missing and the remaining 2 can recreate that which is missing. This is at the cost of 1/3 data capacity utilization, because the parity data is essentially useless on its own.

In the mirrored parity configuration (outlined in the “6 Drive Mirrored Parity” diagram), capacity utilization is halved (from 66.7% to 33.3%), however any 3 pieces of data out of the 6 can be lost and the lost data can still be recovered. Here are some examples:

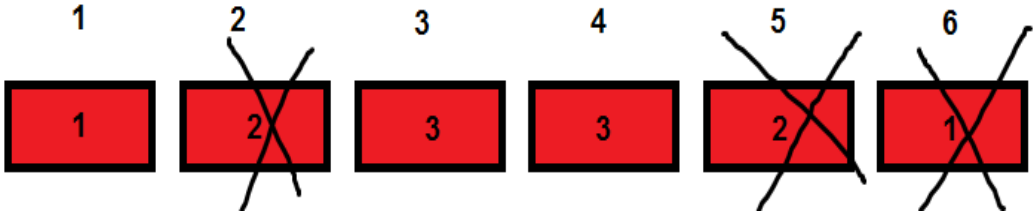


6 Drive Mirrored Parity, missing Drives 2, 3, and 4



6 Drive Mirrored Parity, missing Drives 1, 3, and 5

These examples are able recover because 1 and 2 are still intact (“AB” and “CD” from the example). The following example is able recover its missing segment 2, because an XOR operation can be performed with the remaining segments 1 and 3 to recover the missing data:



6 Drive Mirrored Parity, missing Drives 2, 5, and 6

Any combination of 3 drives can fail in this configuration. Even some combinations of 4 drive failures can recover. There are 15 possible combinations of 4 drive failures for this configuration.

The technique for determining all possible combinations of 4 drives of failure involved finding all 6-bit binary numbers that contain 4 bits which are 1 (leaving 2 zero bits). The 1 bits represent a drive failure and the 0 bits indicate the drive is still intact. To do this, I created software (using the C# language, utilizing the for loop construct) to loop from 0 up to the maximum number that a 6-bit number can store (2^6 or 64). Here is an example of the loop:

```
for (int j = 0; j < Math.Pow(2, totalDrives); ++j)
{
        Additional code here...
}
```

Each number in this loop was passed to a function that determined the quantity of bits which were 1 through performing a recursive AND gate logical operation. This process began with the number from the current iteration of the loop, performed the AND operation against the number minus 1, then assigned the result back to number itself, until the number was equal to 0. The number of AND operations required until this number is equal zero is the number of 1 bits in the binary representation of the original number. Here is the code for the function (in C#):

```
public static short CountOneBits(int number)
{
    short count = 0;
    for (; (number != 0); number &= (number - 1))
      ++count;
    return count;
}
```

The AND gate returns a high output (1) only if both inputs are high. Otherwise, it produces a low output (0).

2 Input AND Gate		
Input A	Input B	Output
0	0	0
0	1	0
1	0	0
1	1	1

For an example of this function in action, let's say I passed in 45. It would first perform an AND operation on 45 with itself minus 1 (44). 45 in binary is 00101101 and 44 in binary is 00101100.

45 AND 44		
45	44	Output
0	0	0
0	0	0
1	1	1
0	0	0
1	1	1
1	1	1
0	0	0
1	0	0

The result is 44. Next, an AND operation is performed on 44 with itself minus 1 (43). 44 in binary is 00101100 and 43 in binary is 00101011.

44 AND 43		
44	43	Output
0	0	0
0	0	0
1	1	1
0	0	0
1	1	1
1	0	0
0	1	0
0	1	0

The result is 28. Next, an AND operation on 28 with itself minus 1 (27). 28 in binary is 00101000 and 27 in binary is 00011011.

28 AND 27		
28	27	Output
0	0	0
0	0	0
1	0	0
0	1	0
1	1	1
0	0	0
0	1	0
0	1	0

This result is 8. Finally, an AND operation on 8 with itself minus 1 (7). 8 in binary is 00001000 and 7 in binary is 00000111.

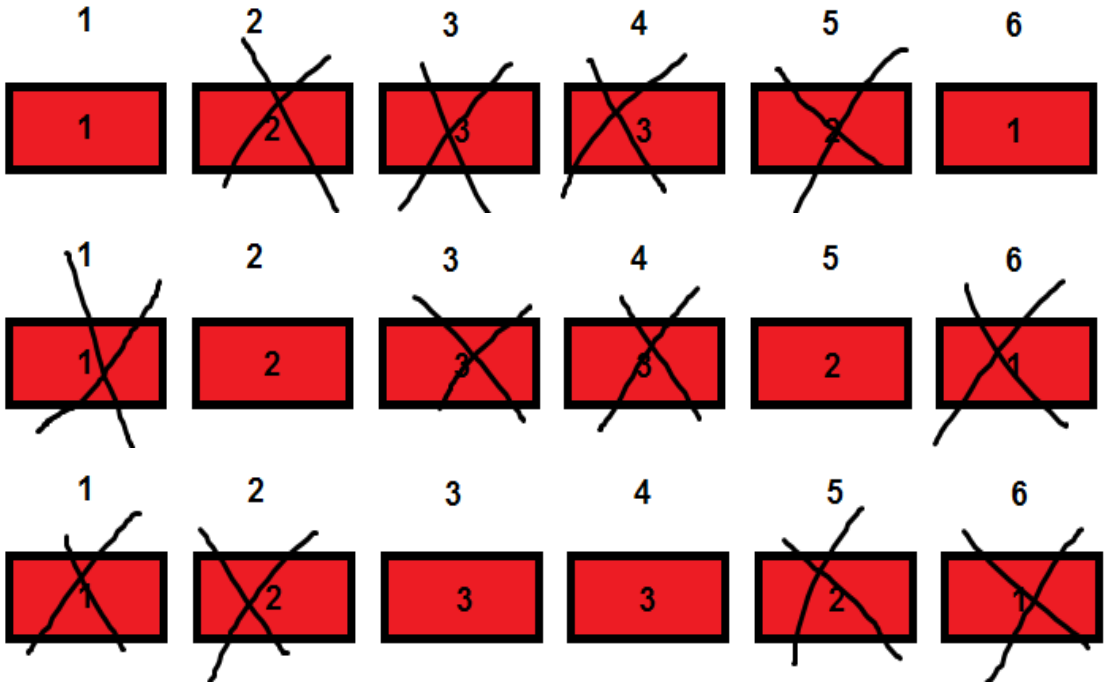
8 AND 7		
8	7	Output
0	0	0
0	0	0
0	0	0
0	0	0
1	0	0
0	1	0
0	1	0
0	1	0

The result is 0. This took a total of 4 AND operations, which is equal to the number of one bits in our original number 45.

To determine 4 drives of failure, if the number returned by the bit counting function was 4, then the number was utilized, otherwise the calling method would continue its loop until it reached the next number for the desired occurrence of 1 bits, or its maximum limit. The implementation would convert the utilized numbers into a string of its binary representation, then iterate through this string to simulate the drive failure combinations.

I created software to determine every possible combination of multiple drives of failure, until the maximum drive failure count was reached for both drive configurations, using the previously mentioned bit counting technique. This software was written as a C# Console Application and is titled “Failure Report Generator”. It utilizes a technique to determine if an unrecoverable failure has occurred through analyzing the drive configuration’s design and determining which sections remain after simulating each failure combination. A particular drive failure combination marks the configuration as failed if one or more pieces of its data set are unrecoverable. The program outputs the results of the combinations of multiple drive failures to a log file, one failure combination per line, with failed combinations marked “BAD”.

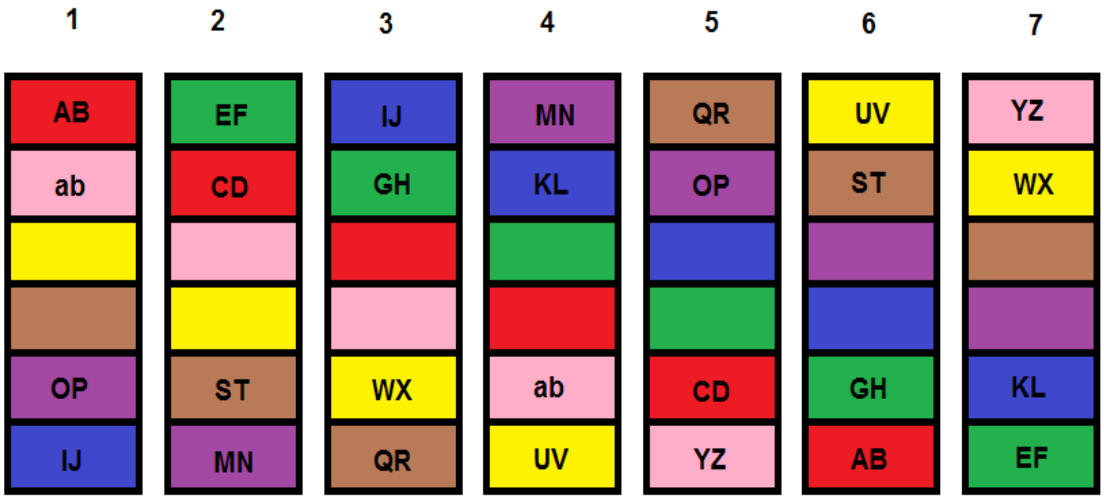
12 of the 15 possible combinations of 4 drive failure for the 6 drive configuration are able to recover. The following are the 3 combinations of 4 drive failure which cannot recover:



The 3 possibilities of 4 drives of failure not possible in 6 drive configuration

Notice that there is only 1 piece remaining out of its 3 unique pieces of data in these failure combinations.

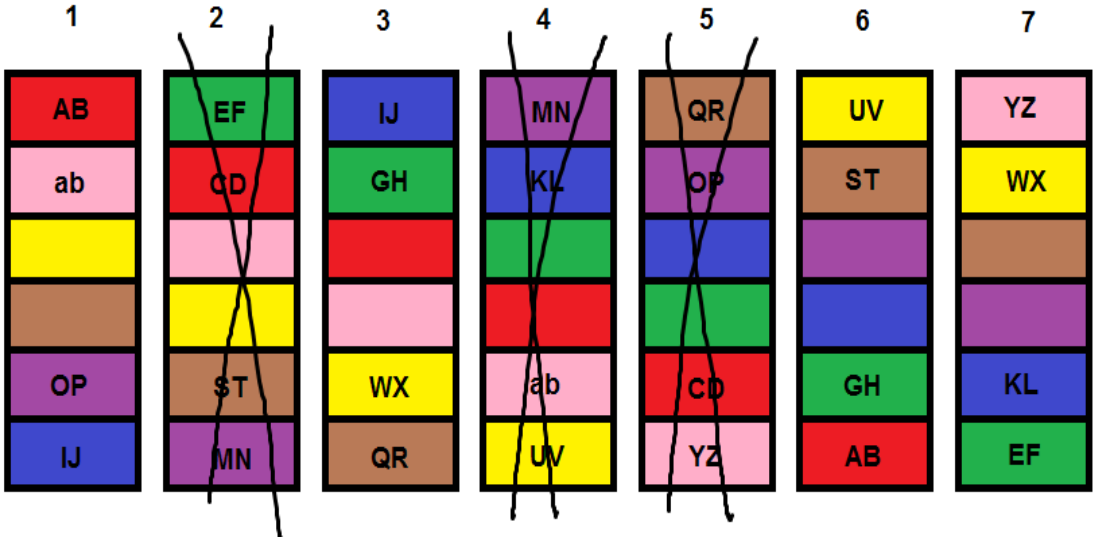
The “7 Drive Dispersed Interleaved Mirrored Parity” configuration has a relatively more complex design. It utilizes similar concepts, except divides and disperses data differently. It uses 7 data sets instead of 1. For example, the text “ABCDEFGHIJKLMNOPQRSTUVWXYZab” would be dispersed as follows:



7 Drive Dispersed Interleaved Mirrored Parity, example of text dispersion

In this example, each color represents a data set, and the unlabeled blocks are the parity for the corresponding color. For example, the unlabeled green blocks are the parity of “EF” and “GH”. Note that this configuration by design will not offer more data capacity, as this example may lead you to think; the example is intended to illustrate the data dispersion technique. Actual implementations of both configurations would offer the same data utilization, which is 33.3%. This is because the parity takes up 1/3, leaving 2/3 for data, however this 2/3 for data is divided in half because the data is mirrored, which comes out to 33.3%.

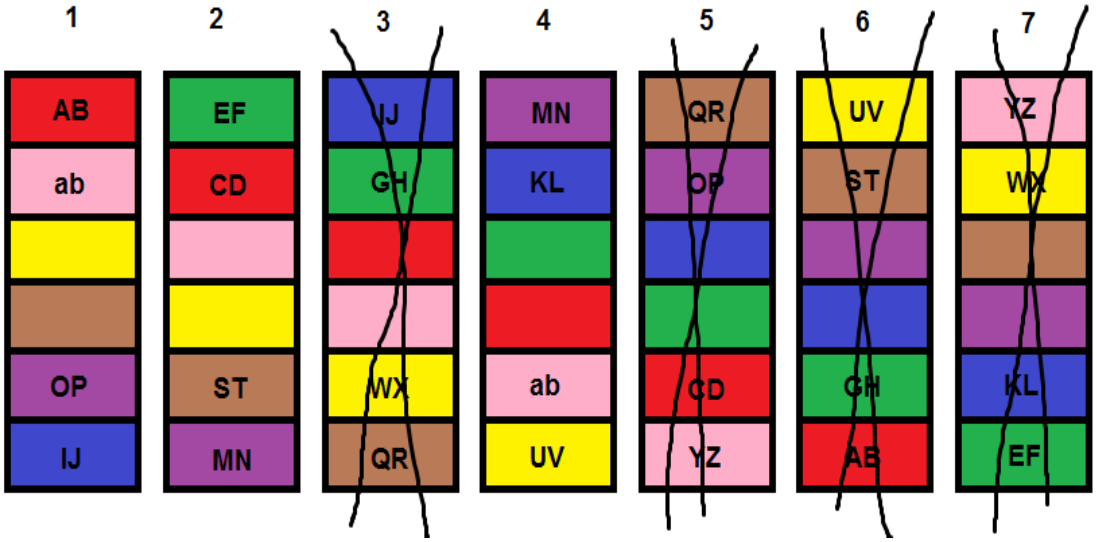
This 7 drive configuration offers 3 drives of failure of all possibilities of 3 drive failure, same as the 6 drive configuration. For example:



7 Drive Dispersed Interleaved Mirrored Parity, example of 3 drives of failure

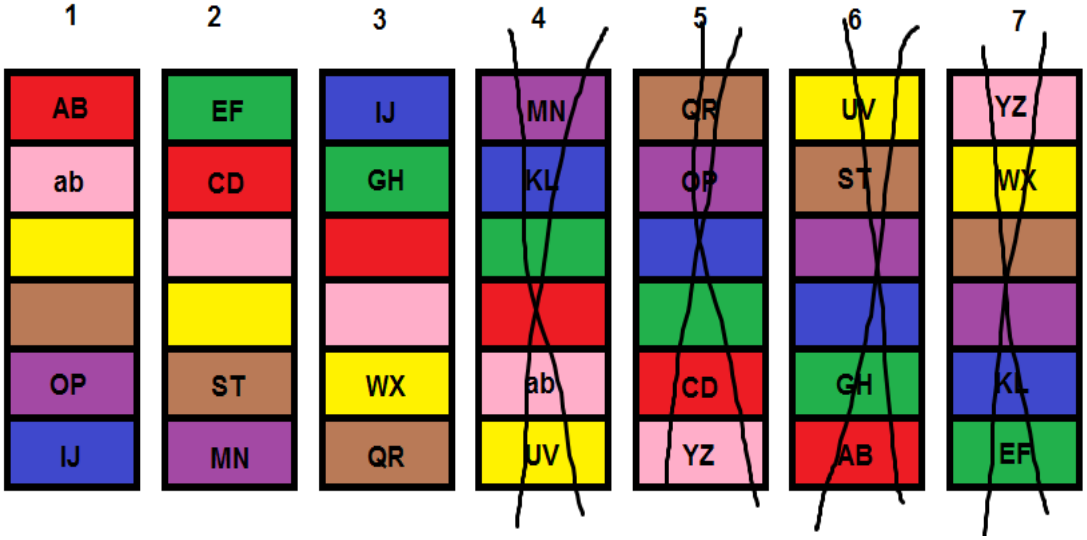
In this situation, "AB" is present in the red of drives 1 and 6, and "CD" is lost from red on drives 2 and 5. Here, "CD" can be recovered because the parity is still intact from the red of drive 3. If you follow this pattern, you will find that "MN" is also recoverable, and it is also possible to regenerate any missing parity segments.

Similar to the 6 drive configuration, there are certain situations permissive of 4 drives of failure. The following is an example of an occurrence:



7 Drive Dispersed Interleaved Mirrored Parity, example of recoverable 4 drive failure

Utilizing the analytical approach previously outlined, there is either both data pieces remaining, or one piece with a corresponding parity, for each set. However, this is not the case for the following occurrence of 4 drives of failure:



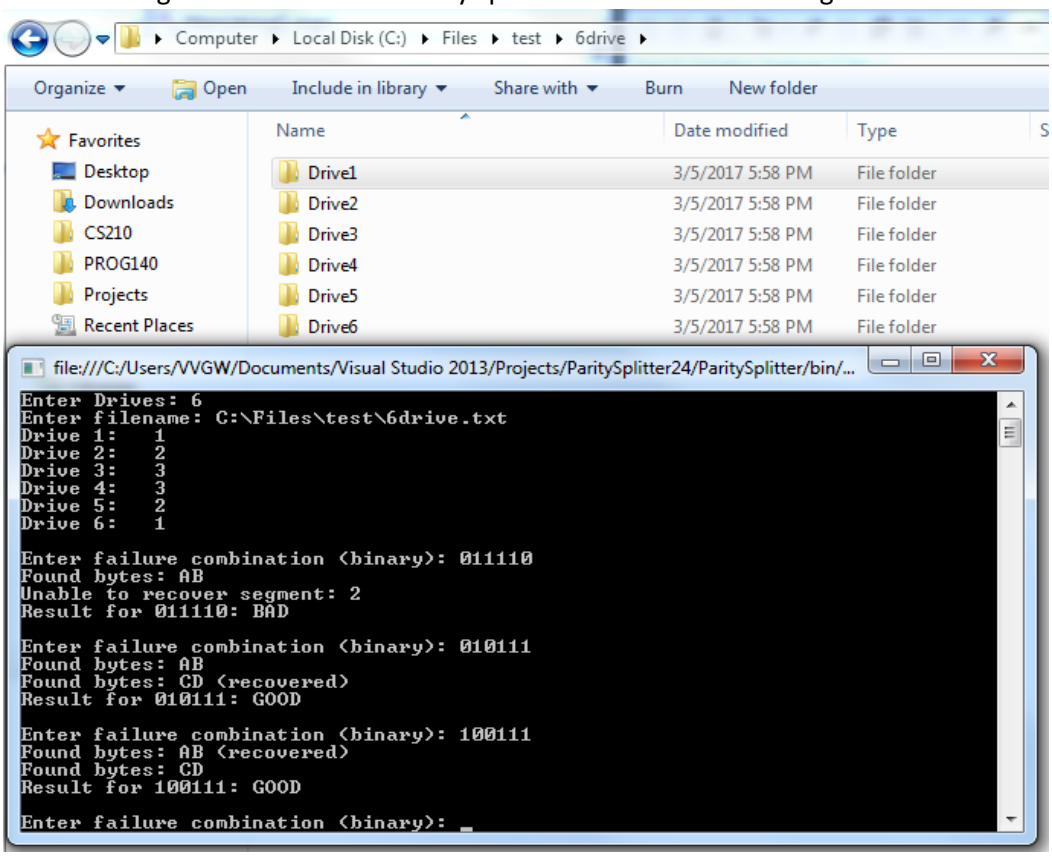
7 Drive Dispersed Interleaved Mirrored Parity, example of an unrecoverable 4 drive failure

Here, every color is fine except for blue, which contains only 1 of its 3 pieces of data ("IJ"), preventing recovery of "KL". This missing/unrecoverable segment marks the configuration unrecoverable.

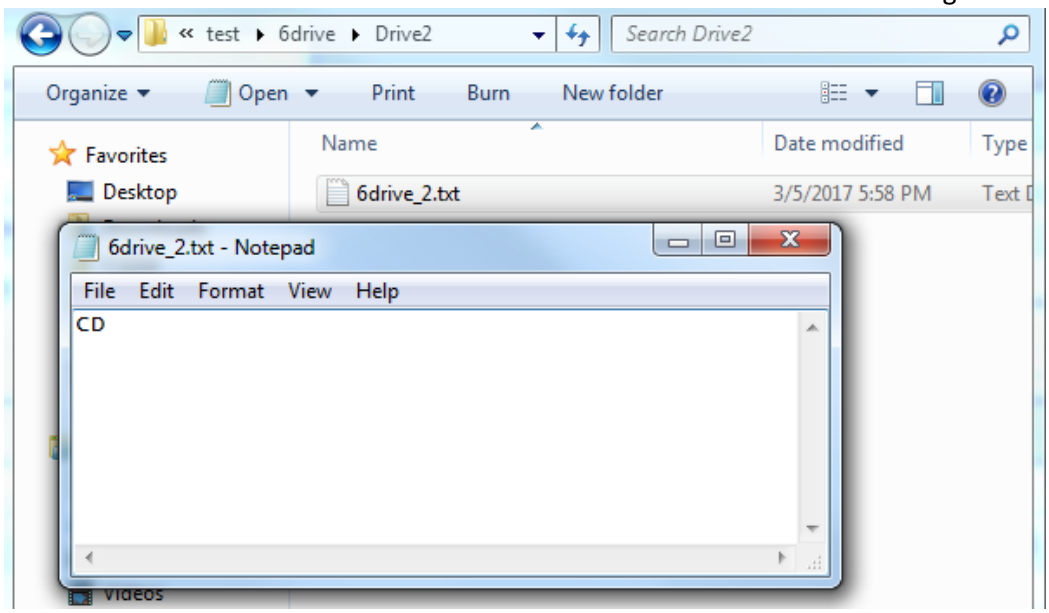
The same bit counting technique, utilized for the 6 drive configuration, was also used to generate all possibilities of 4 drive failure for the 7 drive configuration. However, this time the loop iterates up to the maximum value a 7-bit number can store (2^7 or 128). This results in a total of 35 unique combinations that the 7 drive configuration can experience 4 drives of failure.

I created software to test each combination of failure for data recovery ability. This software was written as a C# Console Application and is titled "Parity Splitter". It begins by prompting the user for the number of drives, then asks for a text file to split. The text file is split into its corresponding sections, then stored in ram and written to the hard drive, under the file's directory with a new subdirectory titled (number of drives) + "drive", then a separate directory is created for each drive. The text file sections are stored in these drive folders, named with the file's name along with an underscore, the segment number, and the same extension as the file (typically ".txt"). Next, the program prompts for a binary failure combination, with the left-most bit representing the first drive and the right-most bit representing the highest numbered drive. 0 represents that the drive has not failed, and 1 represents the drive has failed. The program iterates through each character of the user input to simulate each drive failure and ensures all data is recoverable, using the data from each remaining segment (unlike the drive configuration analysis technique utilized by the "Failure Report Generator" software). I tested every combination of 4 drive failure from the log file generated by the "Failure Report Generator" program, for both drive configurations.

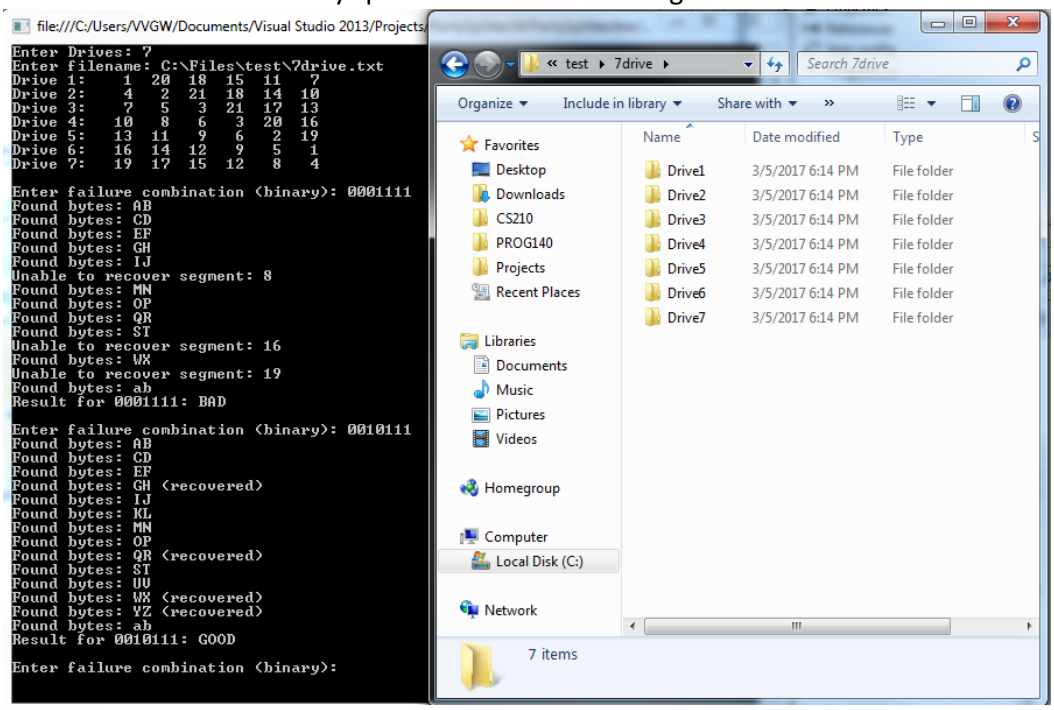
The following is a screenshot of “Parity Splitter” for the 6 drive configuration:



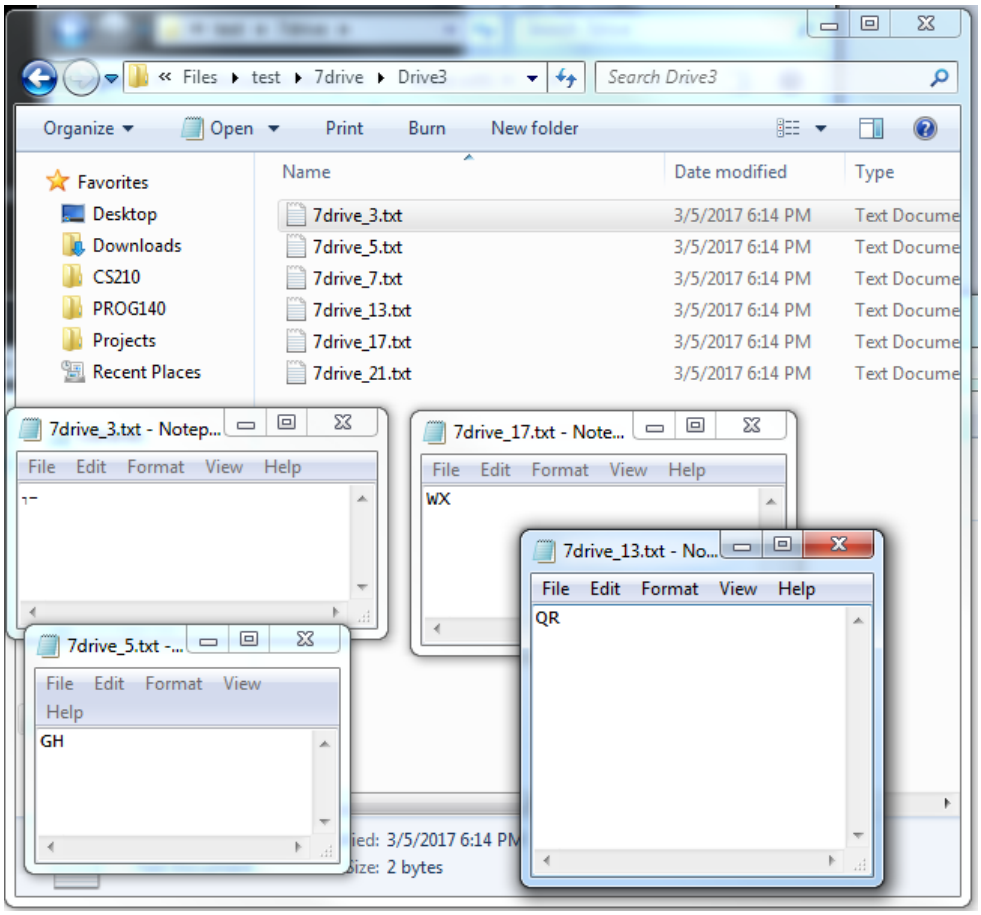
Here is a screenshot of the contents of the “Drive2” folder for the 6 drive configuration:



Here is screenshot of “Parity Splitter” for a 7 drive configuration:



Here is a screenshot of the contents of the “Drive3” folder for a 7:

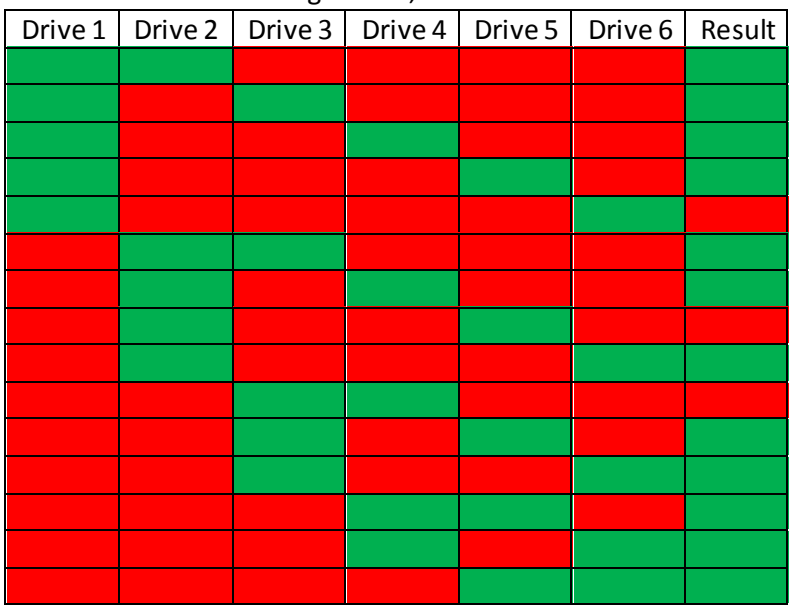


3. Data

Tests were performed on all 15 unique possible combinations of 4 drives of failure for the 6 drive configuration as well all 35 unique possible combinations for the 7 drive configuration to experience 4 drives of failure.

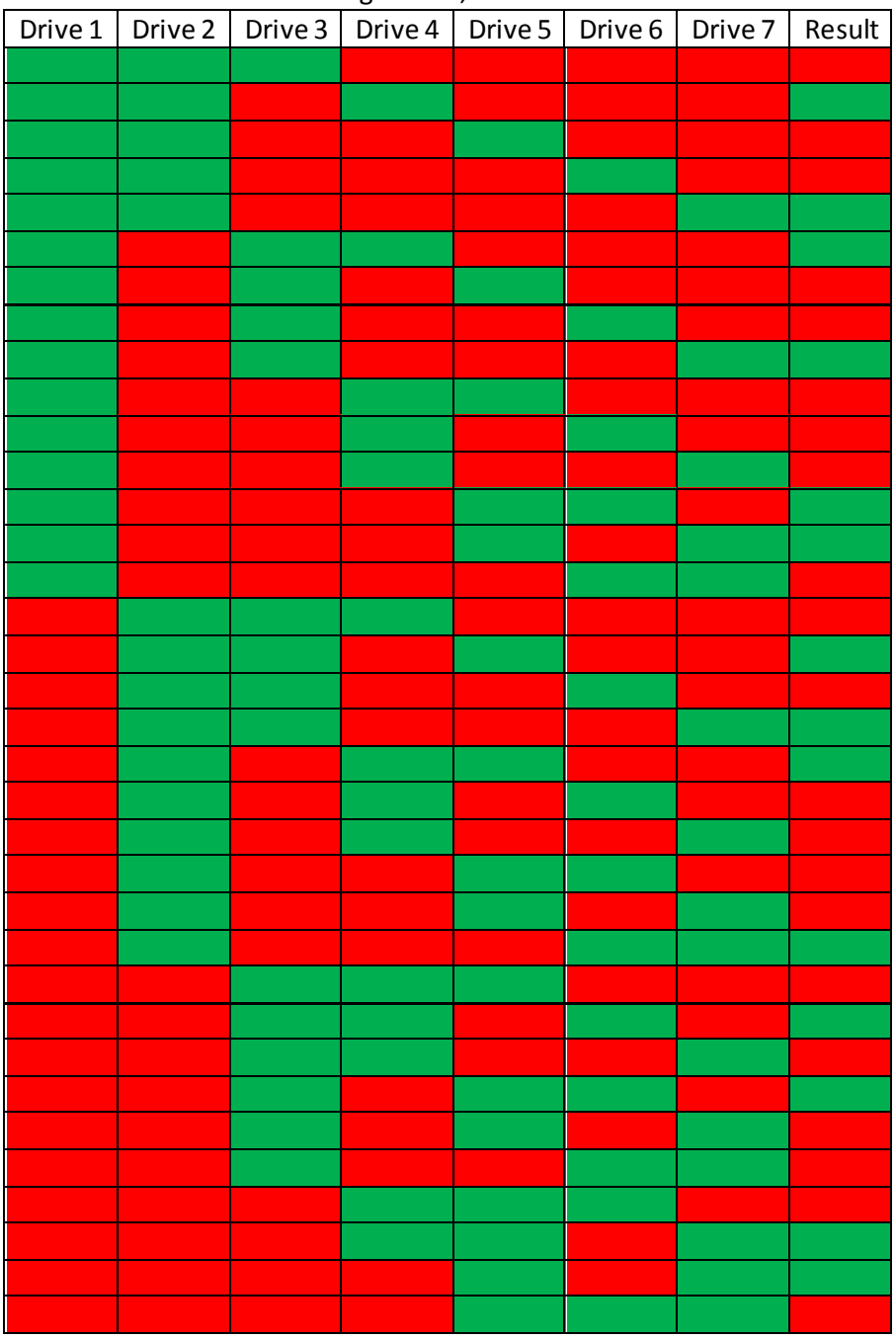
A drive is considered unrecoverable if one or more pieces of its data are unrecoverable, regardless if other pieces of the drive's data are still present or recoverable. This depiction of failure is indicated by a result of red. A result of green indicates the configuration has the ability to recover all of its data. For the specific drives, red represents the individual drive failure, and green signifies the drive is still operational, with its corresponding data intact.

6 drive configuration, 4 drives of failure:



3 out of 15 failures. Failure rate: 20%. Success Rate: 80%

7 drive configuration, 4 drives of failure:



21 out of 35 failures. Failure rate: 60%. Success Rate: 40%

4. Conclusion

My hypothesis proved to be incorrect. The 7 drive configuration did not offer a higher chance at surviving 4 drives of failure over the 6 drive configuration. This means I reject my hypothesis and accept the null hypothesis of the 6 drive configuration offering an equal or higher chance surviving 4 drives of failure over the 7 drive configuration. In fact, the 7 drive configuration was 3 times less likely to survive 4 drives of failure, with a failure rate of 60% versus the 6 drive configuration's 20% failure rate.

The basis for my hypothesis came from my understanding that the extra drive (on the 7 drive configuration) would allow for more possibilities for 4 drives of failure. While this was the case, interleaving the data likely created more possibilities for failure combination conflicts. However, further analysis is required to investigate the issue.

My motivation for developing this design was to find a superior data redundancy solution over what is currently available to the public. Upon researching the issue, I discovered that there are already superior methods. "Reed-Solomon error correction", developed in the 60's, is utilized today in CDs, DVDs, Blue Ray Discs, QR codes, RAID configurations, and more. It allows for the creation of multiple parities using a mathematical algorithm and offers better data utilization compared to the drive configurations presented in this paper.

I did develop "Failure Report Generator" and "Parity Splitter" beyond the scope of this paper, as I was forming my hypothesis. For instance, they were designed to allow for higher numbered drive configurations than 7; they will actually allow up to 32 drives. It should be noted that data capacity increases with these higher drive configurations (at a rate beyond the scope of this paper). Around the 30 drive point, the computing power required, along with the size of the log files generated by the "Failure Report Generator" software, become unmanageable, so I terminated the program. These log files indicated that there are combinations of higher numbers of drive failure than 5 (starting at 8 and 11 drives, respectively). I included some of the logs generated by the "Failure Report Generator" software (in the zip file submitted with this report) if you are interested in looking at this data, along with additional diagrams and the source code and executables for the "Failure Report Generator" and "Parity Splitter" software.