Task Description

In this assignment you will be reading and interpreting binary data from a VR headset. You will need to open the binary file containing the headset data, read the data points in chunks and process each chunk. The name of the binary file will be provided as the first command line argument. The next three command line arguments represent individual bytes that form a 3-byte delimiter where the delimiter separates the stream of binary data into chunks.

Before attempting this assignment it would be a good idea to familiarise yourself with pointers and pointer arithmetic, binary files, file I/O and C strings. In particular you may find the following functions helpful:

```
// File I/O
fopen(), fclose(), fread(), fwrite(), fseek(), ftell(), feof()

// C Strings and Pointers
strtol(), memcmp(), memset(), memcpy()
```

You will not need dynamic memory allocation for this assignment. You must not make any calls to `malloc()` of `free()` in your implementation.

Some implementation details are purposefully left ambiguous; you have the freedom to decide on the specifics yourself. Additionally this description does not define all possible behaviour that can be exhibited by the system; some error cases are not documented. You are expected to gracefully report and handle these errors yourself.

Introduction

In Linux, everything is a file, including hardware devices such as keyboards, mouses or controllers. Communication with hardware devices is done through streams of bytes rather than text to minimise the amount of data transferred. To do this efficiently, both the sender and receiver of the data have to agree to a protocol which allows for a more efficient encoding of the information. The sender encodes their information according to the protocol as a stream of bytes. Then they send the raw bytes to the receiver and the receiver uses the same protocol to interpret the data stream and decode the original information. When sending raw bytes, it is common that a sequence of bytes is intended to be processed together as a single chunk. To ensure that the receiver knows that a particular sequence of bytes is to be processed as one chunk, each chunk is separated by a special sequence of bytes known as the delimiter. By relying on delimiters, the receiver can know that every byte that it has received since the last delimiter (or the beginning of the communication) forms the current chunk.

One complication can occur when the data stream is sent wirelessly, where it gets corrupted due to interference with other wireless signals. To defend against these errors and ensure that the receiver can detect when a corruption occurs, the sender sends the data (also called the payload) with a checksum of the payload appended to the message. One of the simplest checksums to compute is called a parity bit, where given a sequence of bits in the payload, i.e. 0101, a parity bit is appended to the message to indicate whether there were an even or odd number of 1 bits in the payload. In this case, a 0 would be appended to the payload and the entire message would be sent as 01010. Using a parity bit for the checksum protects against an odd number of bit flips in the message but fails to protect against an even number of bit flips. Although this encoding doesn't protect against every type of corruption, it is simple and works well in practice.

In this assignment, you will read the contents of a binary file representing the data received by a VR headset using a protocol. The first step will be to isolate the stream of bytes into chunks by searching for the delimiter. Each chunk is a sequence of packets, each representing the headset's X, Y and Z coordinates, a swizzle operation and a checksum. Each packet will be checked for validity and processed as part of that chunk.

Compilation and Execution

Your program should be written in a C file named vr2017.c, and it will then be compiled by running the default rule of a make file. Upon compiling, your program should produce a single 'vr2017' binary. Your binary should accept four command-line arguments in the form of the path to a binary file to execute, followed by the three bytes of the delimiter.

For example:

```
./vr2017 01.bin 0x33 0x4F 0xFA
```

The above command provides the name of the binary file as the first command-line argument "01.bin," followed by the three bytes of the delimiter each represented as hexadecimal strings 0x33, 0x4F and 0xFA.

Failing to adhere to these conventions will prevent your markers from running your code and tests. In this circumstance, you will be awarded a mark of 0 for this assignment.

Example: 1 Chunk 1 Packet

Suppose that the program is invoked as follows:

```
./vr2017 01.bin 0xFF 0xAA 0x11
```

First, the program should perform validation on its command-line arguments in the following order:

1. Check that it has been invoked with the correct number of command line arguments.
2. Check that the path to the file it has been provided refers to a file that actually exists.
3. Check that each delimiter byte is a valid hexadecimal string with two hex digits (either lowercase or uppercase).

Next, the program should print to stdout the decimal value for each delimiter byte. In the above example, the program should print:

```
Delimiter byte 0 is: 255
Delimiter byte 1 is: 170
Delimiter byte 2 is: 17
```

To protect against errors, the delimiter triplet is always sent with a corresponding parity byte. To compute the parity byte for the delimiter triplet, the bytes are written in their binary representation, and then for each of the 8 columns, a parity bit is computed, counting whether there were an even or odd number of 1 bits in that column.

In above example we would write:

```
Delimiter byte 0: 11111111
Delimiter byte 1: 10101010
Delimiter byte 2: 00010001
                  --------
     Parity byte: 01000100
```

After computing the parity byte, it should be printed to stdout as follows:

```
Checksum: 68
```

Consider an example where the contents of 01.bin are as follows:

```
0x7B 0xDE 0x71 0x02 0xD6 0xFF 0xAA 0x11 0x44
```

We can visualise the binary file using a diagram.

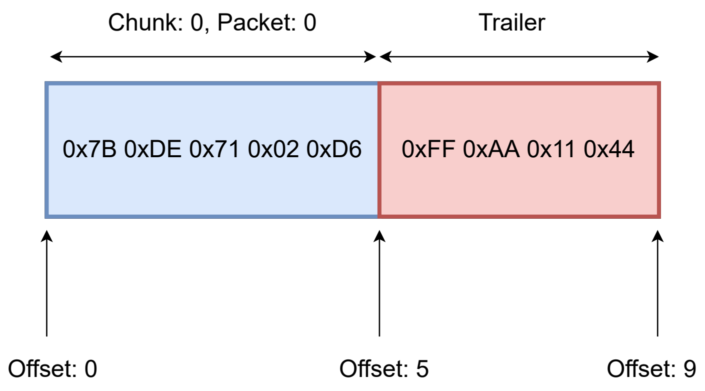

Figure 1: Contents of 01.bin

Each binary file represents a series of chunks (shown in blue), separated by trailers (shown in red). Each chunk contains zero or more packets, where each packet is exactly 5 bytes. Each trailer is exactly 4 bytes, with the first 3 bytes being the delimiter byte triplet provided in the command line arguments and the last byte being the corresponding checksum.

After processing the command-line arguments, the next step of the program is to scan the binary file for the first trailer. If there is a match for the 4 bytes of the trailer, which in the above example occurs at offset 5, then a chunk has been found. A chunk spans from the previous trailer to the current trailer. If there is no previous trailer, then the beginning of the file (offset 0) is taken as the beginning of the chunk. In our example, since the trailer occurs at offset 5 and there is no previous trailer, the chunk spans from offset 0 to offset 5.

When the chunk is located, we print its index (starting from 0 and incrementing as each new chunk is found) and the offset that indicates the beginning of the chunk. So, for this example, the program should print:

```
Chunk: 0 at offset: 0
```

As each packet in the chunk is processed, the index should be printed to stdout. The packet index starts at 0 at the beginning of a new chunk and increments by 1 for every new packet. The index resets when moving to the next chunk.

```
    Packet: 0
```

Next, the chunk contents can be divided into 5-byte packets. If the size of a chunk is not a multiple of 5, then an error is printed, and the program continues to the next chunk. The maximum number of packets in a chunk is 128, therefore a chunk never exceeds the size of 128 * 5 = 640 bytes.

Each 5 byte packet in a chunk is of the following form:

```
[B0, B1, B2, Swizzle, Checksum]
```

Each field of the packet is a single byte. B0, B1 and B2 represent the coordinates in 3D space of the VR headset before swizzling. The swizzle indicates the order to interpret the bytes B0, B1 and B2, and the checksum is a parity byte of the B0, B1, B2 and swizzle bytes. The first step is to check the packet's validity by computing the parity byte of the B0, B1, B2 and swizzle bytes and comparing that with the appended checksum.

For the above example we would write:

```
     B0 byte: 01111011
     B1 byte: 11011110
     B2 byte: 01110001
Swizzle byte: 00000010
              --------
 Parity byte: 11010110
```

In this case, the provided checksum and the calculated checksum are a match, and we can proceed with processing this packet. On the other hand, if the calculated checksum and provided checksum are not a match, then an error should be printed to stdout, and the program should move on and process the next packet in the chunk.

The next part of the computation is to perform a swizzle on the B0, B1 and B2 bytes according to the value of the swizzle byte. The swizzle byte represents an order to rearrange the provided B0, B1 and B2 bytes into X, Y and Z coordinates. Below is a lookup table that maps the swizzle byte's value to the permutation order.

```
1 -> XYZ
2 -> XZY
3 -> YXZ
4 -> YZX
5 -> ZXY
6 -> ZYX
```

If the swizzle byte is not between 1 and 6, an error should be printed to stdout and the next packet processed. In our example, the swizzle byte is 0x02 or 2 in decimal which represents the XZY swizzle. To ensure that the data was read correctly before swizzling, the program should print the contents it read directly from the packet:

```
Data before swizzle -> B0: 123, B1: 222, B2: 113
Swizzle: XZY
```

Next, it should perform the swizzle. Since we have an XZY swizzle, it represents that B0 is X, the B1 is Z, and B2 is Y. Therefore, after swizzling, the data should be printed to stdout as follows:

```
Data after swizzle -> X: 123, Y: 113, Z: 222
```

The last step of processing each chunk is to print out an average X, average Y and average Z value for the entire chunk. In this case, there was only one packet in the chunk, so the averages correspond to the values we observed in the one packet. The chunk averages should be printed as floating-point numbers with 2 decimal places. So in the above example, it should print:

```
    Chunk Average X: 123.00, Average Y: 113.00, Average Z: 222.00
```

Therefore, for the entire execution of the program with the above command line arguments and binary file, the output to stdout should be:

```
Delimiter byte 0 is: 255
Delimiter byte 1 is: 170
Delimiter byte 2 is: 17
Checksum: 68

Chunk: 0 at offset: 0
    Packet: 0
       Data before swizzle -> B0: 123, B1: 222, B2: 113
       Swizzle: XZY
       Data after swizzle -> X: 123, Y: 113, Z: 222
    Chunk Average X: 123.00, Average Y: 113.00, Average Z: 222.00
```

Example: 2 Chunks 4 Packets

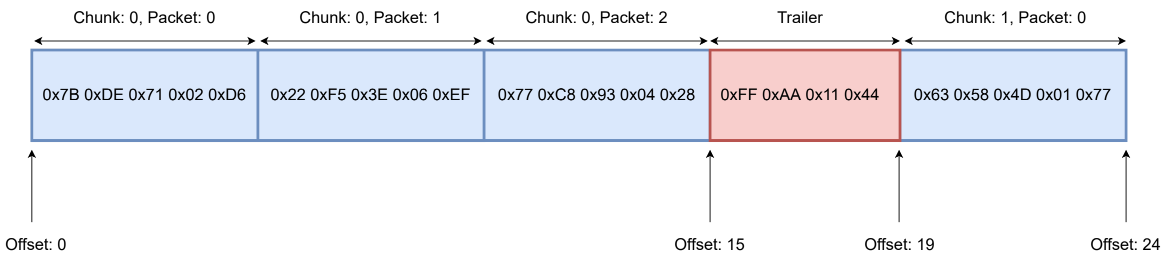

Figure 2: Contents of 02.bin

We consider a slightly more complicated example involving discarding packets with exceptional values and handling remaining packets sent without a trailer.

We execute the program with the following command line arguments:

```
./vr2017 02.bin 0xFF 0xAA 0x11
```

Note that although the file has been changed to 02.bin, the 3-byte delimiter (and hence trailer) remain the same as before.

After finding the trailer at offset 15, the contents from offset 0 to offset 15 are processed as a single chunk. The first packet remains identical to the previous example. Thus after processing the first packet of the first chunk, the output to stdout is as follows:

```
Delimiter byte 0 is: 255
Delimiter byte 1 is: 170
Delimiter byte 2 is: 17
Checksum: 68

Chunk: 0 at offset: 0
    Packet: 0
        Data before swizzle -> B0: 123, B1: 222, B2: 113
        Swizzle: XZY
        Data after swizzle -> X: 123, Y: 113, Z: 222
```

When processing the second packet, we observe values B0: 34, B1: 245, B2: 62, Swizzle: 6 (ZYX). The program should print the values before and after the swizzle just like before:

```
    Packet: 1
        Data before swizzle -> B0: 34, B1: 245, B2: 62
        Swizzle: ZYX
        Data after swizzle -> X: 62, Y: 245, Z: 34
```

However, at this point, another check has to be conducted to make sure that the X, Y and Z values are reasonable. To do this, we check that the difference between the X, Y and Z coordinates of the current packet and the X, Y and Z coordinates of the last valid packet in the chunk do not differ by more than 25. In this case, the X value of the current packet is 62, and the X value of the last valid packet was 123. Therefore, the difference is |62-123| = 61, which exceeds 25. Therefore, this packet is to be discarded and not included in the average for this chunk or any further calculations.

```
    Ignoring packet. X: 62. Previous valid packet's X: 123. 61 > 25.
```

When processing the third packet in the chunk, we observe values B0: 119, B1: 200, B2: 147, Swizzle: 4 (YZX). After swizzling, we have the coordinates X: 147, Y: 119, Z: 200. We compare these values to the previous valid packet in the chunk (Packet 0) and find that all values are reasonable. Therefore, the packet is included in the average.

Therefore, the output for the entire chunk should read:

```
Chunk: 0 at offset: 0
    Packet: 0
        Data before swizzle -> B0: 123, B1: 222, B2: 113
        Swizzle: XZY
        Data after swizzle -> X: 123, Y: 113, Z: 222
    Packet: 1
        Data before swizzle -> B0: 34, B1: 245, B2: 62
        Swizzle: ZYX
        Data after swizzle -> X: 62, Y: 245, Z: 34
        Ignoring packet. X: 62. Previous valid packet's X: 123. 61 > 25.
    Packet 2:
        Data before swizzle -> B0: 119, B1: 200, B2: 147
        Swizzle: YZX
        Data after swizzle -> X: 147, Y: 119, Z: 200
    Chunk Average X: 135.00, Average Y: 116.00, Average Z: 211.00
```

The last point of consideration is processing the stream of bytes at the end of the binary file if no trailer follows it. In the above example, chunk 1 remains containing one packet (shown in blue) without a trailer. For this assignment, the program should process the end of the data stream as a chunk even if no trailer is present after it. In our example, we will treat the remaining data stream as a second chunk to be processed after the first chunk. Thus, when processing the last chunk, the program should print:

```
Chunk: 1 at offset: 19
    Packet: 0
        Data before swizzle -> B0: 99, B1: 88, B2: 77
        Swizzle: XYZ
        Data after swizzle -> X: 99, Y: 88, Z: 77
    Chunk Average X: 99.00, Average Y: 88.00, Average Z: 77.00
```

Marking Criteria

The following is the marking breakdown, each point contributes a portion to the total 10% of the assignment. You will receive a result of zero if your program fails to compile.

Marks are allocated on the basis of:

- Correctness - 8 - Passing automatic test cases, a number of tests will *not* be released or run until after your final submission.
- Manual Marking - 2 - code quality, comments in the code and describing the file reading pattern.
  - Complete the README.md file with your name and unikey. In this file you are to write the file operations you perform for reading and obtaining the data for examples 1 and 2. For each step through the examples, it should specify which byte offset position the file is in, how many bytes are read, where the byte position moves to etc. These should be specified with numerals in short form: e.g. at 0, read 3, at 3, move forward 10, at 13, move back 7, at 6. 500 word limit.

Correctness is based on:

- Standard output of the `vr2017` program execution for a given input.

Additionally marks will be deducted on the basis of:

- Compilation - If your submission does not compile you will receive an automatic mark of zero for this assessment.
- Style - Poor code readability will result in the deduction of up to 2 marks. Your code should be neatly divided between header and source files in appropriate directories, should be commented, contain meaningful variable names, useful indentation, white space and functions should be used appropriately. Please refer to this course's style guide for more details.

Warning: Any attempts to deceive or disrupt the marking system will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not follow the assignment description or if your code is unnecessarily or deliberately obfuscated.

Helpful Hints / Where to Start

- Start by checking the validity of the command line arguments.
- Next, try opening the binary file and printing out all of its contents.
- Next, try finding all the delimiters in the file. Compare this with your expectation How many are there? and what are the positions of each? if either of those mismatches your expectation, find out!
- Creating your own tests make it easier to start. Limit the scope of the problem: Why not choose a simple and fixed delimiter to start with?