Project [AZUR]

[AZUR] Image File Format Specification Version 1

June 2024

Copyright © 2024 Wing-Chern Leung
All rights reserved
This document may be distributed freely, provided it is unaltered.

Email: contact@leungengineering.com
Website: www.leungengineering.com/azur

Contents

List of Figures iii

List of Tables iv

1 Abstract 1

2 Definitions And Abbreviations Used In This Specification 2

2.1 Definitions . . . 2

2.2 Abbreviations . . . 2

3 Concepts And Conventions Used In This Specification 4

3.1 Byte Ordering . . . 4

3.2 Image Origin . . . 4

3.3 Blocks . . . 5

3.4 Block Data Stream . . . 5

4 Encoding The Image 7

4.1 The Magic Numbers . . . 7

4.2 Image Information . . . 7

4.3 Image Data . . . 8

5 Decoding The Image 9

6 Appendix A - Edge Cases 10

6.1 Whole Image Made Of One Colour . . . 10

6.2 Repetitive Alternation Of Pixels . . . 10

7 Appendix B - Sample Code 12

7.1 Encoder . . . 12

7.2 Decoder . . . 18

List of Figures

1 Byte Ordering Used In This Specification 4
2 Image Origin (0) Of An Array Of Pixels 5
3 Splitting The Image Into 3 Blocks 5
4 Block Stream In Pixels 6
5 Block Stream In Bytes 6
6 [AZUR] Image File Format Magic Numbers 7
7 An Image With The Same Colour For Every Pixel 10
8 An Image With Alternating Colour For Every Pixel 10

List of Tables

1 [AZUR] Image Information 7

1 Abstract

This document describes the image file format known as [AZUR], which utilises a lossless compression algorithm that is believed to be original and therefore free from any patents and can thus be practiced and implemented freely.

The goals of this image file format are that it is simple and easy to implement, and that it losslessly compresses cell shaded graphics well, where there are many continuous blocks of repetitive colour, and can therefore be used for High Definition Anime Artwork (HDAA) and any other Computer Graphics (CG) that has a lot of repetitive blocks of colour.

The specification can be used for 32 bit colour and can be used to implement 48 or 64 bit colour in the future.

The essence of the compression method described in this specification is that it utilises Per Pixel Compression (PPC), rather than say, per byte compression, and has a repeat variable to copy pixels to the output. Whilst the pixel value is also a value like any other data, it is important to emphasise that it is the grouping of data into pixels, and the repetition of these pixels, that is the essence of this compression method.

The format uses blocks of the image data to ease the computing effort needed to encode and decode the image.

The recommended filename extension used for this file format is ".azur", as in "hatsune_miku_1.azur".

2 Definitions And Abbreviations Used In This Specification

2.1 Definitions

Alpha: The opacity of a pixel. Zero alpha would mean a totally transparent pixel, whilst maximum alpha represents a completely opaque pixel.

Anime: Japanese influenced animation and artwork.

Aspect Ratio: The ratio of the image width to the image height, such as the 16:9 for High Definition (HD) resolutions.

Bitdepth: The amount of values per pixel component. For example a bitdepth of 8 would mean a value of 0 to 255 inclusive for each component.

Byte: 8 bits; an unsigned value of 0 to 255 inclusive.

Cell Shading: Shading an image with blocks of colour that has sharp edges.

Component: For a RGBA pixel, R would be the red component, G would be the green component, B would be the blue component and A would be the alpha component.

Computer Graphics (CG): Digital (data created, stored, processed, transferred etc. on a computer) images.

Decoding: Transformation of an [AZUR] file into the original image.

Encoding: Transformation of image data into the [AZUR] file format.

High Definition (HD) Image: An image with a resolution equal to or greater than 1280 x 720 pixels (Width x Height).

Lossless Compression Decompression: Encoding and decoding, respectively, of an image without any loss or alteration of image data.

Pixel: A discrete unit of a display monitor, which can have red, green, blue and alpha values.

Raw Bitmap: An image where the image data has been encoded literally. For example, 4 bytes per pixel, for every pixel, for a 32 bit image.

Render: Producing an array of pixels.

Resolution: Amount of pixels that make up the image.

Unsigned Value: A value that is zero or greater, hence the value does not need a negative sign (-).

2.2 Abbreviations

CG: Computer Graphics.

HD: High Definition.

HDAA: High Definition Anime Artwork.

LSB: Least Significant Byte.

MSB: Most Significant Byte.

PPC: Per Pixel Compression.

RGBA: Red, Green, Blue, Alpha.

3 Concepts And Conventions Used In This Specification

3.1 Byte Ordering

For [AZUR] files, the order of multi-byte values are stored with the Most Significant Byte (MSB) first.

For example, a 16 bit value of 4104 has its bits arranged like so:

```
Most Significant Byte (MSB)
Bit Number: 16    15    14    13    12    11    10    9
Value:      32768 16384 8192  4096  2048  1024  512   256
Bits set:   0     0     0     1     0     0     0     0

Least Significant Byte (LSB)
Bit Number: 8     7     6     5     4     3     2     1
Value:      128   64    32    16    8     4     2     1
Bits set:   0     0     0     0     1     0     0     0


                    0001 0000 | 0000 1000
                    MSB-------| LSB------
                    0x10 (16) | 0x08 (8)
```

Figure 1: Byte Ordering Used In This Specification

The 16 bit number is split into two bytes, the Most Significant Byte (MSB) and the Least Significant Byte (LSB). Here we can see that the calculated value is 256 * 16 = 4096 for the MSB and 1 * 8 = 8 for the LSB. Adding these two bytes gives us the number 4104.

When writing multi-byte values to file, the MSB is written first, and subsequent bytes of this value are written left to right in descending order of significance. When reading a multi-byte value from file, the MSB is read first, and subsequent bytes of the value are read from left to right in descending order of significance. This is assuming that the data is written and read similar to reading and writing English on a page of very wide paper.

3.2 Image Origin

An image is a square or rectangular array of pixels that has been rendered on the display. The origin, that is where the dimensions X and Y, and also the pixel locations, start from, is the top left of the image for [AZUR] image files.

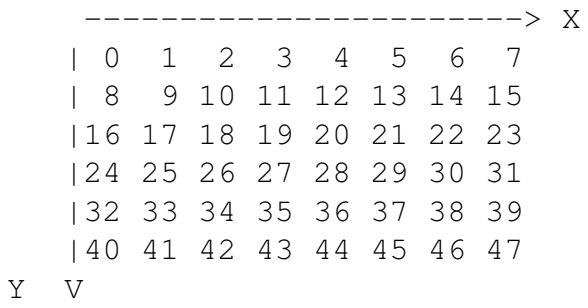

Figure 2: Image Origin (0) Of An Array Of Pixels

The X coordinates increase from left to right and the Y coordinates increase from top to bottom.

3.3 Blocks

Consider the array of pixels in the figure above. If it is split into n blocks, then it will have n amount of image sections with the height of each section or block equal to the image height/n. The image width will remain unchanged. In this example, n = 3.

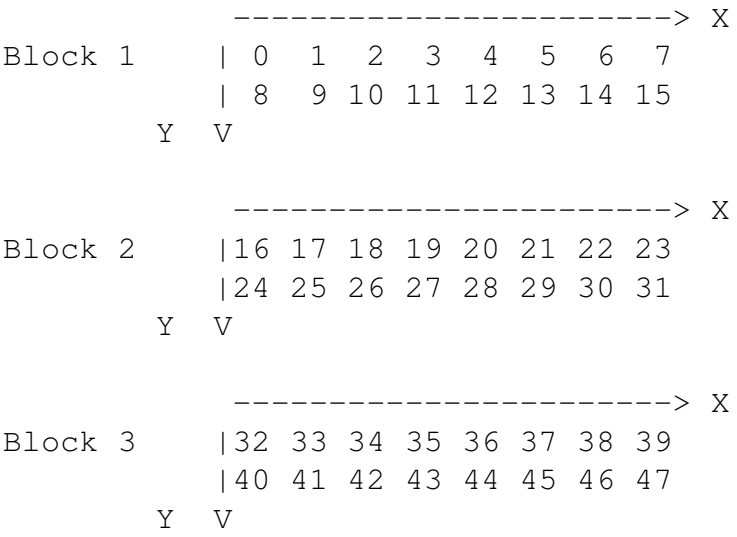

Figure 3: Splitting The Image Into 3 Blocks

The image height must be divisible by the amount of blocks with no remainder, and the amount of blocks must be an integer greater than or equal to 1. In other words: image height % n blocks is equal to 0.

3.4 Block Data Stream

The block data stream, or just block stream, is an array of a number of bytes equal to the image width multiplied by the block height multiplied by the amount of bytes per pixel. For example, the block Block 1 in the figure above has a corresponding block stream assuming an RGBA bitdepth of 8:

```
Pixel: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Figure 4: Block Stream In Pixels

```
Byte: 0R 0G 0B 0A 1R 1G 1B 1A 2R 2G 2B 2A... 15R 15G 15B 15A
```

Figure 5: Block Stream In Bytes

The block stream is an array intended to be used to process the data for each block, and can therefore be reused for each block.

4 Encoding The Image

4.1 The Magic Numbers

Every compliant [AZUR] image must start with a 4 byte magic number:

```
0x41 0x5A 0x55 0x52
A    Z    U    R
```

Figure 6: [AZUR] Image File Format Magic Numbers

The character A is the MSB and the bytes decrease in order of significance from left to right until the LSB character R.

4.2 Image Information

Following the magic numbers are the image information fields. They must be written to the image file in the order top to bottom of the following table:

Order	Amount of bytes	Information Field
1	1	File Version Major
2	1	File Version Minor
3	1	File Version Micro
4	4	Image Width In Pixels
5	4	Image Height In Pixels
6	1	Bitdepth
7	1	Compression Method
8	4	n Blocks

Table 1: [AZUR] Image Information

To be compliant with this specification, the File Version Major field should be set to 1.

To be compliant with this specification, the File Version Minor field should be set to 0.

To be compliant with this specification, the File Version Micro field should be set to 0.

Whilst no real restriction is imposed on the image width and height, it is recommended to use values that conform to a standard HD resolution or to an aspect ratio equal to a standard HD resolution aspect ratio such as the 16:9.

For 32 bit RGBA colour, the Bitdepth field should be set to 8.

The only compression method described in this specification is Compression Method 0. Therefore, in order to be compliant with this specification, the byte for the Compression Method field must be equal to 0.

The n Blocks field must be an integer that has a value that is greater than or equal to 1, and that will divide the image height with no remainder. In other words, image height % n blocks is equal to 0.

4.3 Image Data

After the image information is the image data; the following describes the compression method Compression Method 0.

The image data, for 32 bit RGBA pixels, consists of a repetition of a 4 byte value for the pixel, that is stored in the order RGBA, followed by a 4 byte repeat that is used to copy repetitions of the pixel to the output. For example:

32 bit RGBA Pixel: 0xFF 0x00 0x00 0xFF
32 bit Repeat: 0x00 0x00 0x00 0x01

will result in two red pixels. An algorithm for the conversion of an image to this format is described below:

Transfer the pixels of the block to the block stream array.

Then note the first pixel of the block stream as the current pixel, and check to the right of the pixel. If it is the same, then increment the repeat counter and repeat as long as the pixel is the same. If the next pixel is different then write the current pixel followed by the repeat.

If the pixel to the right of the first pixel is different, then write the first pixel followed by a repeat of 0.

Then repeat this process for the next pixel, noting the new pixel as the current pixel, for up to the last pixel of the block stream.

On the last pixel of the block stream, if it is the same as the current pixel, then increment the repeat counter and write the current pixel followed by the repeat. If it is different, then write the current pixel followed by the repeat, followed by the last pixel followed by a repeat of 0.

Repeat for each other block.

In other words, the image data of a block, for 32 bit RGBA pixels, is in the form: pixel 1 (4 bytes), repeat 1 (4 bytes), pixel 2 (4 bytes), repeat 2 (4 bytes), pixel 3 (4 bytes), repeat 3 (4 bytes)... and so on until the end of the block.

5 Decoding The Image

The following describes an algorithm for decoding an [AZUR] image file:

Read the magic numbers and check they match.

Read the image information fields.

Check if the image blocks is not 0.

Check if the image height is divisible by n Blocks with no remainder.

For each block:

First copy the RGBA values of the pixel to the block stream and increase the pixel count by 1. Then read the repeat and copy the pixel to the block stream to the amount of the repeat value and increment the pixel count per repetition.

For every pixel in the block stream, copy to the image buffer and when it has filled an entire row (the amount being equal to the image width), start filling in the next row.

Once the entire image has been read in, check the pixel count matches the total pixels calculated from the image width and height.

6 Appendix A - Edge Cases

6.1 Whole Image Made Of One Colour

Consider the image in the following figure, where R represents a 4 byte RGBA pixel of the component values R: 0xFF, G: 0x00, B: 0x00 and A: 0xFF:

```
R R R R R R R R
R R R R R R R R
R R R R R R R R
R R R R R R R R
R R R R R R R R
R R R R R R R R
```

Figure 7: An Image With The Same Colour For Every Pixel

Using Compression Method 0 and 1 block, and after it has been converted to a block stream, the entire image has in theory been encoded in 8 bytes: 4 bytes for the first pixel and 4 bytes for the repeat of 47.

In theory, it will compress better, with respect to a raw bitmap, using RGBA bitdepths greater than 8 and a 32 bit repeat. Also, as the image resolution (amount of pixels) increase, the image data will stay at a size of 8 bytes as long as the repeat is less than or equal to the maximum amount of values possible with a 32 bit data size.

6.2 Repetitive Alternation Of Pixels

Consider the image in the following figure, where R represents a 4 byte RGBA pixel of the component values R: 0xFF, G: 0x00, B: 0x00 and A: 0xFF, and G represents a 4 byte RGBA pixel of the component values R: 0x00, G: 0xFF, B: 0x00 and A: 0xFF:

```
R G R G R G R G
R G R G R G R G
R G R G R G R G
R G R G R G R G
R G R G R G R G
R G R G R G R G
```

Figure 8: An Image With Alternating Colour For Every Pixel

Using Compression Method 0 and 1 block, and after it has been converted to a block stream, the entire image has in theory doubled in size with respect to a raw bitmap: for every pixel there is 4 bytes for the pixel value and 4 bytes for a repeat of 0.

Whilst at first this edge case may not seem to occur in practice as much as the first edge case (Section 6.1), there are many examples of alternation, or even just a different pixel to the previous pixel, such as photographic images or scans from a scanner.

7 Appendix B - Sample Code

7.1 Encoder

```
/************************************************
    Copyright (C) 2023 - 2024 Wing-Chern Leung
    All rights reserved

    www.leungengineering.com/sonico
************************************************/

int export_azur(void)
{
    uint32_t x = 0;
    uint32_t y = 0;
    uint32_t z = 0;

    uint32_t i = 0;
    uint32_t j = 0;
    uint32_t k = 0;

    // Get RGB values from layer data
    uint8_t *rgb;

    // The bitdepth
    uint8_t bitdepth = 8;

    // n blocks
    uint32_t blocks = 8;

    // Error check blocks
    if(EXPORT_HEIGHT % blocks != 0)
    {
        fprintf(stderr, "Please use a number of blocks that divides the
        export height without a remainder\n");

        return 1;
    }

    if(blocks < 1)
    {
        fprintf(stderr, "Please use an amount of blocks greater than or
```

```
    equal to 1\n");

    return 2;
}

uint32_t block_height = EXPORT_HEIGHT/blocks;

uint32_t block_bytes = EXPORT_WIDTH * block_height * 4 * (bitdepth/8);

// Image matrix into block_stream bytes
uint8_t block_stream[block_bytes];

// If new pixel not the same as current pixel or end of block
uint8_t new_pixel = 1;

// RGBA of current pixel (can be 64 bit)
uint8_t current_pixel[8];
// Repeat counter
uint32_t repeat = 0;

// Scale the image to the export size
export_resample(EXPORT_WIDTH, EXPORT_HEIGHT);

// Open the file for writing
fp_azur_export = fopen(file_name_azur_export, "w");

/*********************
    Magic numbers
*********************/
// 4 bytes magic (AZUR)
putc(0x41, fp_azur_export); // A
putc(0x5A, fp_azur_export); // Z
putc(0x55, fp_azur_export); // U
putc(0x52, fp_azur_export); // R

/************************
    Image information
************************/
// File Version Major
putc(0x00, fp_azur_export); // Major: 0
// File Version Minor
putc(0x00, fp_azur_export); // Minor: 0
```

```
// File Version Micro
putc(0x01, fp_azur_export); // Micro: 1
// Image width in pixels (4 bytes)
hex_bytes = int_to_hex(EXPORT_WIDTH);
putc(hex_bytes[3], fp_azur_export);
putc(hex_bytes[2], fp_azur_export);
putc(hex_bytes[1], fp_azur_export);
putc(hex_bytes[0], fp_azur_export);
// Image height in pixels (4 bytes)
hex_bytes = int_to_hex(EXPORT_HEIGHT);
putc(hex_bytes[3], fp_azur_export);
putc(hex_bytes[2], fp_azur_export);
putc(hex_bytes[1], fp_azur_export);
putc(hex_bytes[0], fp_azur_export);
// Bit depth (per RGBA component)
putc(bitdepth, fp_azur_export);     // 8
// Compression method 0
putc(0x00, fp_azur_export);         // 0
// Blocks (4 bytes)
hex_bytes = int_to_hex(blocks);
putc(hex_bytes[3], fp_azur_export);
putc(hex_bytes[2], fp_azur_export);
putc(hex_bytes[1], fp_azur_export);
putc(hex_bytes[0], fp_azur_export);

/*****************
    Image data
*****************/
// Convert matrix[H][W] to block_stream[X]
for(z = 0, y = 0; z < blocks; z++)
{
    // Get a section of image
    for(i = 0, j = 0; i < block_height; i++, y++)
    {
        for(x = 0; x < EXPORT_WIDTH; x++)
        {
            if(alpha_mode == ALPHA_MODE_OFF)
            {
                // Then just get pixel
                rgb = int_to_hex(layer_10_export[y][x]);

                block_stream[j + 0] = rgb[2];   // R
```

```
            block_stream[j + 1] = rgb[1];   // G
            block_stream[j + 2] = rgb[0];   // B
            block_stream[j + 3] = 0xFF;     // A
        }
        else if(alpha_mode == ALPHA_MODE_ON)
        {
            // Then check if 0xFFFFFF (if so then R = 0xFF
            // G = 0xFF, B = 0xFF, A = 0x00)
            rgb = int_to_hex(layer_10_export[y][x]);

            // If white
            if(layer_10_export[y][x] == 0xFFFFFF)
            {
                // Then max transparency
                block_stream[j + 0] = 0xFF;     // R
                block_stream[j + 1] = 0xFF;     // G
                block_stream[j + 2] = 0xFF;     // B
                block_stream[j + 3] = 0x00;     // A
            }
            // Else
            else
            {
                // Just get pixel
                block_stream[j + 0] = rgb[2];   // R
                block_stream[j + 1] = rgb[1];   // G
                block_stream[j + 2] = rgb[0];   // B
                block_stream[j + 3] = 0xFF;     // A
            }
        }

        j += (bitdepth/8) * 4;
    }
}

// Process the block stream up to the last pixel
for(k = 0, repeat = 0; k < (j - 4);)
{
    if(new_pixel == 1)
    {
        // putc(pixel) (4 bytes)
        putc(block_stream[k + 0], fp_azur_export);
        putc(block_stream[k + 1], fp_azur_export);
```

```
        putc(block_stream[k + 2], fp_azur_export);
        putc(block_stream[k + 3], fp_azur_export);

        // current_pixel[4] == new pixel
        current_pixel[0] = block_stream[k + 0];
        current_pixel[1] = block_stream[k + 1];
        current_pixel[2] = block_stream[k + 2];
        current_pixel[3] = block_stream[k + 3];

        k += 4;

        new_pixel = 0;
    }
    else
    {
        // Check to the right of pixel of block_stream
        // Compare next_pixel to current_pixel

        // if(same) then increment count (repeat++) and move right
        // one more
        if(current_pixel[0] == block_stream[k + 0] &&
        current_pixel[1] == block_stream[k + 1] &&
        current_pixel[2] == block_stream[k + 2] &&
        current_pixel[3] == block_stream[k + 3])
        {
            // if(same) then repeat++ and k += 4
            repeat++;
            k += 4;
        }
        // if(not same)
        else
        {
            // putc(repeat)
            hex_bytes = int_to_hex(repeat);
            putc(hex_bytes[3], fp_azur_export);
            putc(hex_bytes[2], fp_azur_export);
            putc(hex_bytes[1], fp_azur_export);
            putc(hex_bytes[0], fp_azur_export);

            // Then repeat = 0, new_pixel = 1
            repeat = 0;
            new_pixel = 1;
```

```
        }
    }
}
// Process the block stream's last pixel
// If the last pixel is the same as the current pixel
if(current_pixel[0] == block_stream[k + 0] &&
current_pixel[1] == block_stream[k + 1] &&
current_pixel[2] == block_stream[k + 2] &&
current_pixel[3] == block_stream[k + 3])
{
    // Increment the repeat counter
    repeat++;

    // putc(repeat)
    hex_bytes = int_to_hex(repeat);
    putc(hex_bytes[3], fp_azur_export);
    putc(hex_bytes[2], fp_azur_export);
    putc(hex_bytes[1], fp_azur_export);
    putc(hex_bytes[0], fp_azur_export);

    // Reset new_pixel for next block
    new_pixel = 1;
}
// Else the last pixel is different from the current pixel
else
{
    // putc(repeat)
    hex_bytes = int_to_hex(repeat);
    putc(hex_bytes[3], fp_azur_export);
    putc(hex_bytes[2], fp_azur_export);
    putc(hex_bytes[1], fp_azur_export);
    putc(hex_bytes[0], fp_azur_export);

    // putc(pixel) (4 bytes)
    putc(block_stream[k + 0], fp_azur_export);
    putc(block_stream[k + 1], fp_azur_export);
    putc(block_stream[k + 2], fp_azur_export);
    putc(block_stream[k + 3], fp_azur_export);

    // putc(repeat) of 0
    repeat = 0;
```

```
            hex_bytes = int_to_hex(repeat);
            putc(hex_bytes[3], fp_azur_export);
            putc(hex_bytes[2], fp_azur_export);
            putc(hex_bytes[1], fp_azur_export);
            putc(hex_bytes[0], fp_azur_export);

            // Reset new_pixel for next block
            new_pixel = 1;
        }
    }

    // Close the file
    fclose(fp_azur_export);

    printf("Drawing exported to: %s\n", file_name_azur_export);

    return 0;
}
```

7.2 Decoder

```
/*****************************************
    Copyright (C) 2024 Wing-Chern Leung
    All rights reserved

    www.leungengineering.com/illustrious
*****************************************/

int file_open(FILE *ifp)
{
    // 4 chars (AZUR)
    uint8_t magic[MAX_MAGIC];

    // File version
    uint8_t version_major = 0;
    uint8_t version_minor = 0;
    uint8_t version_micro = 0;

    // Arrays for converting char to unsigned byte or unsigned integer,
    // respectively
    uint8_t hex_8[1];
    uint8_t hex_32[4];
```

```
// Get char with getc();
uint8_t c = 0;

// Loop count
uint32_t x = 0;
uint32_t y = 0;
uint32_t z = 0;
uint32_t i = 0;
uint32_t j = 0;
uint32_t k = 0;

uint8_t read_rgba[4];

// Calculate rgb int based on uint8_t RGB chars from read in
uint64_t rgba_32 = 0;

uint8_t image_bitdepth = 0;
uint8_t image_compression_method = 0;
uint32_t image_blocks = 0;

uint64_t repeat = 0;

uint32_t total_pixels = 0;

// The block stream
uint32_t block_stream[MAX_BLOCK_PIXELS];

/*****************
    Read Magic
*****************/
c = getc(ifp);
magic[3] = c;
c = getc(ifp);
magic[2] = c;
c = getc(ifp);
magic[1] = c;
c = getc(ifp);
magic[0] = c;

// error check magic
if(error_check_magic(magic) == -1)
```

```
{
    return -1;
}

printf("[AZUR] File\n");

/*******************
    Read Version
*******************/
c = getc(ifp);
hex_8[0] = c;
version_major = hex_8[0];

// error check version_major
if(error_check_uchar(version_major) == -1)
{
    return -1;
}

c = getc(ifp);
hex_8[0] = c;
version_minor = hex_8[0];

// error check version_minor
if(error_check_uchar(version_minor) == -1)
{
    return -1;
}

c = getc(ifp);
hex_8[0] = c;
version_micro = hex_8[0];

// error check version_micro
if(error_check_uchar(version_micro) == -1)
{
    return -1;
}

printf("File Version: ");
printf("%i.%i.%i\n", version_major, version_minor, version_micro);
```

```
/***********************
    Read image width
***********************/
c = getc(ifp);
hex_32[3] = c;
c = getc(ifp);
hex_32[2] = c;
c = getc(ifp);
hex_32[1] = c;
c = getc(ifp);
hex_32[0] = c;

image_width = char_to_int_32(hex_32);

printf("Image Width: %i\n", image_width);

/************************
    Read image height
************************/
c = getc(ifp);
hex_32[3] = c;
c = getc(ifp);
hex_32[2] = c;
c = getc(ifp);
hex_32[1] = c;
c = getc(ifp);
hex_32[0] = c;

image_height = char_to_int_32(hex_32);

printf("Image Height: %i\n", image_height);

/********************
    Read bitdepth
********************/
c = getc(ifp);
hex_8[0] = c;

image_bitdepth = hex_8[0];

printf("Image Bitdepth: %i\n", image_bitdepth);
```

```
/*****************************
    Read compression method
*****************************/
c = getc(ifp);
hex_8[0] = c;

image_compression_method = hex_8[0];

printf("Image Compression Method: %i\n", image_compression_method);

/******************
    Read blocks
******************/
c = getc(ifp);
hex_32[3] = c;
c = getc(ifp);
hex_32[2] = c;
c = getc(ifp);
hex_32[1] = c;
c = getc(ifp);
hex_32[0] = c;

image_blocks = char_to_int_32(hex_32);

printf("Image Blocks: %i\n", image_blocks);

if(image_blocks < 1)
{
    file_open_error = E_BLOCKS_ZERO;
    return -1;
}

if(image_height % image_blocks != 0)
{
    file_open_error = E_BLOCKS;
    return -1;
}

/**********************
    Read image data
**********************/
for(i = 0; i < image_blocks; i++)
```

```
{
    // For each block
    for(j = 0; j < (image_width*image_height/image_blocks);)
    {
        // Get R
        c = getc(ifp);
        hex_8[0] = c;
        read_rgba[2] = hex_8[0];

        // error check R
        if(error_check_uchar(read_rgba[2]) == -1)
        {
            return -1;
        }

        // Get G
        c = getc(ifp);
        hex_8[0] = c;
        read_rgba[1] = hex_8[0];

        // error check G
        if(error_check_uchar(read_rgba[1]) == -1)
        {
            return -1;
        }

        // Get B
        c = getc(ifp);
        hex_8[0] = c;
        read_rgba[0] = hex_8[0];

        // error check B
        if(error_check_uchar(read_rgba[0]) == -1)
        {
            return -1;
        }

        // Get A
        c = getc(ifp);
        hex_8[0] = c;
        read_rgba[3] = hex_8[0];
```

```
    // error check A
    if(error_check_uchar(read_rgba[3]) == -1)
    {
        return -1;
    }

    rgba_32 = char_to_int_32(read_rgba);

    block_stream[j] = rgba_32;
    j++;

    c = getc(ifp);
    hex_32[3] = c;
    c = getc(ifp);
    hex_32[2] = c;
    c = getc(ifp);
    hex_32[1] = c;
    c = getc(ifp);
    hex_32[0] = c;

    repeat = char_to_int_32(hex_32);

    for(k = 0; k < repeat; k++, j++)
    {
        block_stream[j] = rgba_32;
    }
}

// Copy block_stream[x] to layer_data[y][x]
for(z = 0, x = 0; z < j; z++)
{
    layer_data[y][x][0] = block_stream[z];

    x++;

    if(x == image_width)
    {
        x = 0;
        y++;
    }
}
```

```
        total_pixels += j;
    }

    printf("Total Pixels: %i\n", total_pixels);

    // Check if total pixels calculated == total pixels read
    if(total_pixels != image_width * image_height)
    {
        file_open_error = E_PIXELS;
        return -1;
    }

    return 0;
}
```