Dept. of Electronics and Electrical Communications Engineering
Indian Institute of Technology, Kharagpur

Image and Video Processing Laboratory

[EC69211]

EXPERIMENT – 2

Read, Write and Manipulate BMP image files.

Name : Karthik, Shivam Maji

Roll No. : 22EC39017, 22EC39029

Date of submission : 20-08-2025

1 Aim

The primary objectives of this experiment are:

1. **BMP File Reading:** Implement a modular function to read BMP image files and extract complete header information including height, width, bit depth, file size, and pixel data offset.
2. **BMP File Writing:** Develop a function to write image pixel data back to disk in valid BMP format while preserving all header information and file structure integrity.
3. **Color Channel Manipulation:** Create functions to manipulate individual color channels (Red, Green, Blue) by setting specific channels to zero while maintaining the intensity values of other channels.
4. **Format Compatibility:** Ensure all functions support multiple BMP formats including 24-bit RGB, 8-bit grayscale, and 8-bit color indexed images.
5. **Low-Level Understanding:** Gain deep understanding of bitmap file structure, binary data manipulation, and pixel-level image processing without relying on high-level image processing libraries.

2 Theory

2.1 BMP File Format Structure

The BMP (Bitmap) file format is a raster graphics image file format used to store bitmap digital images. Unlike compressed formats like JPEG or PNG, BMP files store raw pixel data with minimal compression, making them ideal for understanding fundamental image structure.

2.1.1 File Header Structure

The BMP file consists of several distinct sections:

Table 1: BMP File Header Structure (14 bytes)

Offset	Size	Field	Description
0	2	Signature	File type identifier ("BM")
2	4	File Size	Total file size in bytes
6	4	Reserved	Reserved fields (usually 0)
10	4	Data Offset	Offset to pixel data

2.1.2 Info Header Structure

Table 2: BMP Info Header Structure (40 bytes)

Offset	Size	Field	Description
14	4	Header Size	Size of info header (40 bytes)
18	4	Width	Image width in pixels
22	4	Height	Image height in pixels
26	2	Planes	Number of color planes (1)
28	2	Bits Per Pixel	Color depth (1, 4, 8, 24, 32)
30	4	Compression	Compression method
34	4	Image Size	Size of pixel data
38	4	X Resolution	Horizontal resolution
42	4	Y Resolution	Vertical resolution
46	4	Colors Used	Number of colors in palette
50	4	Important Colors	Number of important colors

2.1.3 Color Table (for indexed images)

For images with bit depths 8, a color palette follows the info header. Each entry contains:

- Blue component (1 byte)
- Green component (1 byte)
- Red component (1 byte)
- Reserved/Alpha (1 byte)

2.1.4 Pixel Data Organization

- Pixels are stored row by row, bottom to top
- Each row is padded to 4-byte boundaries
- 24-bit pixels use BGR (Blue-Green-Red) order
- 8-bit pixels reference color table indices

2.2 Row Padding Calculation

BMP format requires each row to be padded to 4-byte (32-bit) boundaries:

$$\text{Row Padding} = (4 - (\text{Width} \times \frac{\text{Bits Per Pixel}}{8}) \bmod 4) \bmod 4 \quad (1)$$

3 Methodology

3.1 Object-Oriented Design Approach

We implemented a comprehensive `Image` class that encapsulates all BMP processing functionality. This design provides several advantages:

- **Encapsulation:** All image data and metadata stored as instance attributes
- **Automatic Processing:** File reading occurs during object initialization
- **State Preservation:** Original data maintained through backup/restore patterns
- **Code Reusability:** Methods can be called multiple times on same image

3.2 Implementation Strategy

3.2.1 Class Architecture

```
class Image():
    def __init__(self, filename):
        self.read_bmp_image(filename)

    def read_bmp_image(self, filename):
        # Parse BMP file structure

    def writeBMP(self, filename):
        # Write BMP file with proper formatting

    def has_color_table(self):
        # Check if image uses indexed colors

    def remove_red(self, filename):
        # Remove red color channel

    def remove_green(self, filename):
        # Remove green color channel

    def remove_blue(self, filename):
        # Remove blue color channel
```

Listing 1: Image Class Structure

4 Implementation

4.1 BMP File Reading

The core reading function parses the BMP file structure sequentially:

```
def read_bmp_image(self, filename):
    with open(filename, "rb") as f:
        # Validate BMP signature
        self.signature = f.read(2).decode("ascii")
        if self.signature != "BM":
            raise ValueError("Not a valid BMP file.")

        # Read file header
        self.file_size = int.from_bytes(f.read(4), "little")
        self.reserved = int.from_bytes(f.read(4), "little")
        self.data_offset = int.from_bytes(f.read(4), "little")

        # Read info header
        self.info_header_size = int.from_bytes(f.read(4), "little")
        self.width = int.from_bytes(f.read(4), "little")
        self.height = int.from_bytes(f.read(4), "little")
        self.planes = int.from_bytes(f.read(2), "little")
        self.bits_per_pixel = int.from_bytes(f.read(2), "little")
        self.compression = int.from_bytes(f.read(4), "little")
        self.image_size = int.from_bytes(f.read(4), "little")
        self.XpixelsperM = int.from_bytes(f.read(4), "little")
        self.YpixelsperM = int.from_bytes(f.read(4), "little")
        self.colors_used = int.from_bytes(f.read(4), "little")
        self.imp_colors = int.from_bytes(f.read(4), "little")
```

Listing 2: BMP File Reading Implementation

4.2 Color Table Processing

For indexed color images (8 bits), we extract the color palette:

```
if self.bits_per_pixel <= 8:
    if self.colors_used == 0:
        self.colors_used = 2 ** self.bits_per_pixel
    self.color_table = [[], [], [], []]  # [Blue, Green, Red, Reserved]
    for _ in range(self.colors_used):
        blue = int.from_bytes(f.read(1), "little")
        green = int.from_bytes(f.read(1), "little")
        red = int.from_bytes(f.read(1), "little")
        reserved = int.from_bytes(f.read(1), "little")
        self.color_table[0].append(blue)
        self.color_table[1].append(green)
        self.color_table[2].append(red)
        self.color_table[3].append(reserved)
```

Listing 3: Color Table Processing

4.3 Pixel Data Extraction

The pixel reading process handles both indexed and direct color formats:

```
self.image_array = []
row_padding = (4 - (self.width * self.bits_per_pixel // 8) % 4) % 4

for _ in range(self.height):
    row = []
    for _ in range(self.width):
        if self.bits_per_pixel == 8:
            # Read palette index
            row.append(int.from_bytes(f.read(1), "little"))
        elif self.bits_per_pixel == 24:
            # Read BGR and convert to RGB
            blue = int.from_bytes(f.read(1), "little")
            green = int.from_bytes(f.read(1), "little")
            red = int.from_bytes(f.read(1), "little")
            pixel = (red << 16) | (green << 8) | blue
            row.append(pixel)
    f.read(row_padding)  # Skip padding bytes
    self.image_array.append(row)
```

Listing 4: Pixel Data Reading with Padding

4.4 Color Channel Manipulation

We implemented separate channel removal functions that handle both indexed and direct color formats:

```
def remove_red(self, filename):
    if self.has_color_table():
        # For indexed images: modify color table
        backup = self.color_table[2][:]
        self.color_table[2] = [0 for _ in self.color_table[2]]
        self.writeBMP(filename)
        self.color_table[2] = backup
    elif self.bits_per_pixel == 24:
        # For 24-bit images: modify pixel data
        modified = []
        for row in self.image_array:
            new_row = []
            for pixel in row:
                blue = pixel & 0xFF
                green = (pixel >> 8) & 0xFF
```

```
                new_pixel = (green << 8) | blue  # Remove red
                new_row.append(new_pixel)
            modified.append(new_row)
        backup = self.image_array
        self.image_array = modified
        self.writeBMP(filename)
        self.image_array = backup
```

Listing 5: Red Channel Removal Implementation

5 Results and Observations

5.1 Test Images Analysis

We processed three test images with different characteristics:

Table 3: Test Image Properties

Image	Dimensions	Bit Depth	File Size	Colors Used
cameraman.bmp	256×256	8	66,614 bytes	256
corn.bmp	256×256	24	196,662 bytes	0
pepper.bmp	512×512	24	786,486 bytes	0

5.2 Unique BMP Format Observations

During implementation, we discovered several unique characteristics of BMP files compared to modern image formats:

5.2.1 1. Bottom-Up Pixel Storage

Observation: Unlike most image formats that store pixels top-to-bottom, BMP stores pixels bottom-to-top.

Impact: This requires careful handling when displaying or processing images to avoid vertical flipping.

Code Evidence:

```
# BMP stores rows from bottom to top
# Row 0 in file = Bottom row of image
# Row (height-1) in file = Top row of image
```

5.2.2 2. Mandatory 4-Byte Row Padding

Observation: Every row must be padded to 4-byte boundaries, regardless of actual pixel data size.

Example Calculation:

$$\text{Width} = 257 \text{ pixels} \tag{2}$$

$$\text{Bits per pixel} = 24 \tag{3}$$

$$\text{Bytes per row} = 257 \times 3 = 771 \text{ bytes} \tag{4}$$

$$\text{Padding needed} = (4 - 771 \bmod 4) \bmod 4 = 1 \text{ byte} \tag{5}$$

$$\text{Total row size} = 771 + 1 = 772 \text{ bytes} \tag{6}$$

5.2.3 3. BGR vs RGB Color Order

Observation: BMP uses BGR (Blue-Green-Red) byte order instead of the more common RGB.

Impact: Direct memory copying from BMP to display buffers results in color channel swapping.

Conversion Required:

```
# BMP file: [Blue][Green][Red]
# Convert to: [Red][Green][Blue] for processing
pixel = (red << 16) | (green << 8) | blue
```

5.2.4 5. Color Table Placement

Observation: Color tables are mandatory for 8-bit images but completely absent for >8-bit images.
File Structure Comparison:

Table 4: BMP File Structure by Bit Depth

Section	8-bit Images	24-bit Images
File Header	14 bytes	14 bytes
Info Header	40 bytes	40 bytes
Color Table	1024 bytes (256×4)	0 bytes
Pixel Data	Width×Height bytes	Width×Height×3 bytes

5.3 Color Channel Manipulation Results

5.3.1 Visual Effects Analysis

Table 5: Color Channel Removal Effects

Channel Removed	Resulting Color Cast	Color Theory Explanation
Red	Cyan/Blue-Green	Cyan = Green + Blue
Green	Magenta/Purple	Magenta = Red + Blue
Blue	Yellow/Orange	Yellow = Red + Green

5.3.2 Indexed vs Direct Color Processing

8-bit Indexed Images (cameraman.bmp):

- Color manipulation achieved by modifying palette entries
- Extremely efficient: O(C) where C = number of colors (256)
- All pixels using same palette color affected simultaneously
- File size unchanged after processing

24-bit Direct Color Images (corn.bmp, pepper.bmp):

- Color manipulation requires individual pixel processing
- Time complexity: O(W×H) where W=width, H=height
- Each pixel processed independently
- Bit manipulation operations required for channel extraction

6 Error Handling and Edge Cases

6.1 Implemented Validations

1. **File Signature Validation:** Ensures file begins with "BM"
2. **Bit Depth Support:** Handles 8-bit and 24-bit formats

3. **Row Padding Calculation:** Proper 4-byte alignment
4. **Color Table Detection:** Automatic indexed color handling
5. **File I/O Error Handling:** Graceful failure management

6.2 Edge Cases Encountered

6.2.1 1. Zero Colors Used Field

Issue: Some 8-bit BMP files set `colors_used = 0`
Solution: Automatically calculate as $2^{\text{bits\_per\_pixel}}$

```
if self.colors_used == 0:
    self.colors_used = 2 ** self.bits_per_pixel
```

6.2.2 2. Odd Width Images

Issue: Images with odd widths require careful padding calculation
Example: 255-pixel wide, 24-bit image:

$$\text{Bytes per row} = 255 \times 3 = 765 \tag{7}$$

$$\text{Padding} = (4 - 765 \bmod 4) \bmod 4 = 3 \text{ bytes} \tag{8}$$

6.3 Bit Manipulation Techniques

6.3.1 Color Component Extraction

```
# Extract components from 24-bit pixel
blue  = pixel & 0xFF           # Get bits 0-7
green = (pixel >> 8) & 0xFF    # Get bits 8-15
red   = (pixel >> 16) & 0xFF   # Get bits 16-23

# Pack components into pixel
pixel = (red << 16) | (green << 8) | blue
```

Listing 6: Efficient Bit Operations

6.3.2 Performance Benefits

- Bitwise operations: O(1) per pixel
- No floating-point arithmetic required
- CPU cache-friendly memory access patterns
- Minimal memory allocations

6.4 Processing Pipeline

```
if __name__ == "__main__":
    images = ["cameraman.bmp", "corn.bmp", "pepper.bmp"]

    for img_path in images:
        print(f"Processing: {img_path}")
        img = Image(img_path)

        # Create backup copy
        img.writeBMP(img_path.replace(".bmp", "_copy.bmp"))
```

```
10
11          # Generate channel variations
12          img.remove_red(img_path.replace(".bmp", "_no_red.bmp"))
13          img.remove_green(img_path.replace(".bmp", "_no_green.bmp"))
14          img.remove_blue(img_path.replace(".bmp", "_no_blue.bmp"))
```

Listing 7: Batch Processing Implementation