RLE-DIB Format Specification

Vito Sartori, 2026

Abstract

This document describes the binary format used by the `RLEDrawToDIB` rendering function identified at address `0x703A2290` of `uxcore.dll`, distributed with Windows Live Messenger 2009. The format stores one or more palette-or-truecolor images as a packed bitstream, optionally arranged as an N-slice (nine-patch) grid, and is consumed directly by a Windows themed-controls renderer driven by DirectUI, a UI framework used internally by Microsoft on Windows Live and Office products. This specification is derived entirely from static analysis of the disassembled binar and verified against a 516-file oracle corpus with pixel-perfect accuracy. It is intended to be sufficient for a clean-room implementation of a parser and PNG exporter on any platform.

Contents

1 Introduction

RLE-DIB is a compact, bitstream-based image format designed to store Windows themed-control artwork. Its salient characteristics are:

- A 6-octet file header encoding image count, grid topology, and design DPI.
- Support for N-slice / nine-patch grid layouts (1x1 through 3x3) to allow resolution-independent border rendering.
- Per-row RLE with four token types (solid fill, delta-up, delta-down, literal) operating on a shared bit-cursor across all colour channels.
- Two precision modes: 5-bit (`values * 8`, range `0-248`) and 8-bit (raw values, range `0-255`).
- Optional per-pixel alpha via a meta-decoder that wraps value tokens with transparent/opaque/explicit-alpha run control.
- Palette mode where decoded channel values serve as indices into an external colour table.

The format is consumed by the function RLEDrawToDIB, which draws directly into a caller-supplied `DIBSECTION` (Device-Independent Bitmap Section).

2 Legal & Research Disclaimer

This material is intended strictly for academic research, reverse engineering for interoperability, and software preservation.

The author does not distribute, reproduce, or provide any proprietary software, including components originally developed by Microsoft.

This work is independent, clean-room in nature, and is not affiliated with, endorsed by, or supported by any original vendor.

Any trademarks, formats, or software referenced remain the property of their respective owners.

3 Conventions and Terminology

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, NOT RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in RFC2119 and RFC8174 when, and only when, they appear in all capitals.

- **Octet** A unit of 8 bits
- **Little-endian** Multi-octet integers stored with least-significant octet first.
- `bits[H:L]` Inclusive bit-field, bit L is the least significant. `bit[N]` is shorthand for `bits[N:N]`.
- `pixel_ptr` A pointer into a sub-image payload, positioned at the start of pixel data (i.e., byte offset +4 from the start of the payload, immediately after the header `DWORD`)
- `run_count` The decoded `run_count` field from a token. Pixel count for that run is `run_count + 1`.
- `B` Per-image run-length bit-width; see Section 9.4.
- `run_ptr_width` The width in octets of each entry in the per-image row-navigation index; see Section 8.1.3.

All multi-octet integer fields are unsigned unless otherwise stated.

4 File Structure Overview

```
+------------------+
| File Header      |  6 octets
+------------------+
| Image Record 0   |  [size_field_width octets: payload_size]
|   Sub-Image Data |  [payload_size octets]
+------------------+
| Image Record 1   |
|   Sub-Image Data |
+------------------+
|       ...        |
+------------------+
| Image Record N-1 |
|   Sub-Image Data |
+------------------+
```

`N = image_count * sub_images_per_set`
`sub_images_per_set` is derived from the `tile_type` nibble (Section 6).

5 File Header (6 octets)

5.1 Magic Octets

Octets 0-2 MUST be `0x52, 0x4C, 0x45` (ASCII `"RLE"`).
A parser MUST reject any input whose first three octets do not match this value.

5.2 Image-Count Octet

Octet 3:

- `bits[6:0]` represents the `image_count` field, containing the number of logical images (1–127);
- `bit[7]` is reserved and MUST be ignored by parsers.

`image_count` MUST be at least 1. A value of 0 is invalid.

5.3 Format-Flags Octet

Octet 4:

- `bits[1:0]` contains the `size_field_width_minus1` field.
- `bits[5:2]` contains the `tile_type` field.
- `bits[7:6]` is reserved and MUST be ignored by parsers.

5.3.1 `size_field_width`

The `size_field_width` is defined as:
`size_field_width = (size_field_width_minus1) + 1`

Valid range is 1–4 octets. This is the width of the payload-size field in every image record (Section 7.1).

5.3.2 `tile_type` Nibble

A 4-bit value in the range `0x0--0xF`; see Section 6 for full semantics.

5.4 DPI-Flags Octet

Octet 5:

- `bits[2:0]` contains the `dpi_code` field, indicating the design DPI code (see table below);
- `bits[7:3]` is reserved and MUST be ignored by parsers.

dpi_code	Design DPI	Scaling factor (at 96 dpi screen)
0	96	1.00 ×
1	120	1.25 ×
2	144	1.50 ×
3	168	1.75 ×
4	192	2.00 ×
5	216	2.25 ×
6	240	2.50 ×
7	–	Not scalable

Table 1: Design DPI encoding

The `design_dpi` value is obtained by applying the formula $(dpi\_code + 4) * 24$ (when `dpi_code` is between 0–6).

6 Tile-Type Semantics

6.1 Sub-Image Count

The `tile_type` nibble is a bitfield:

- `bit[0]` represents the `left_w` component, indicating a fixed-width left column;

- `bit[1]` represents the `top_h` component, indicating a fixed-height top row;
- `bit[2]` represents the `right_w` component, indicating a fixed-width right column;
- `bit[3]` represents the `bottom_h` component, indicating a fixed-height bottom row.

The number of sub-images per logical image is derived as follows:

$$\begin{aligned} \texttt{h\_slices} &= \text{popcount}\left(\{\texttt{left\_w}, \texttt{right\_w}\}\right) \in \{0,1,2\} \\ \texttt{v\_slices} &= \text{popcount}\left(\{\texttt{top\_h}, \texttt{bottom\_h}\}\right) \in \{0,1,2\} \\ \texttt{sub\_images\_per\_set} &= (\texttt{h\_slices}+1)\,(\texttt{v\_slices}+1). \end{aligned}$$

Applying this to all possible nibble values, we have:

Nibble	Sub-images	Grid	Flags set
0x0	1	1x1	(none)
0x1	2	2x1	left_w
0x2	2	1x2	top_h
0x3	4	2x2	left_w, top_h
0x4	2	2x1	right_w
0x5	3	3x1	left_w, right_w
0x6	4	2x2	top_h, right_w
0x7	6	3x2	left_w, top_h, right_w
0x8	2	1x2	bottom_h
0x9	4	2x2	left_w, bottom_h
0xA	3	1x3	top_h, bottom_h
0xB	6	2x3	left_w, top_h, bottom_h
0xC	4	2x2	right_w, bottom_h
0xD	6	3x2	left_w, right_w, bottom_h
0xE	6	2x3	top_h, right_w, bottom_h
0xF	9	3x3	left_w, top_h, right_w, bottom_h

Table 2: Nibble encoding for sub-image grid and flags

6.2 Grid Layout

Sub-images within a set are stored in row-major order (left to right, then top to bottom). For a 3x3 (nibble=0xF) nine-patch the order is:

Index	Position
0	Top-left corner
1	Top edge (centre column)
2	Top-right corner
3	Left edge (middle row)
4	Centre
5	Right edge (middle row)
6	Bottom-left corner
7	Bottom edge (centre column)
8	Bottom-right corner

Table 3: Index to grid position mapping

6.3 Fixed-Border Extraction

When rendering a set with `tile_type != 0`, a renderer MUST extract the four fixed-border dimensions before splitting the destination rectangle. These are read from the `native_width` / `native_height` of specific sub-images within the set:

$$\text{If bit[0] (left\_w) is set:} \quad \text{left\_width} = \text{width}\big(\text{sub\_image[0]}\big)$$

$$\text{If bit[1] (top\_h) is set:} \quad \text{top\_height} = \text{height}\big(\text{sub\_image[0]}\big)$$

$$\text{If bit[2] (right\_w) is set:} \quad \text{right\_width} = \text{width}\big(\text{sub\_image[last]}\big)$$

$$\text{If bit[3] (bottom\_h) is set:} \quad \text{bottom\_height} = \text{height}\big(\text{sub\_image[last]}\big)$$

7 Image-Record Table

7.1 Record Format

Every record has the following structure:

```
+----------------------------------------+
| payload_size  (size_field_width octets)|
+----------------------------------------+
| sub-image data (payload_size octets)   |
+----------------------------------------+
```

`payload_size` is an unsigned integer of exactly `size_field_width` octets, stored little-endian. It counts the octets that follow (i.e., it does NOT include itself).

7.1.1 Record Ordering

Records are written in the following order:

Algorithm 1 Record emission

```
for i ← 0 to image_count − 1 do
    for p ← 0 to sub_images_per_set − 1 do
        write record for logical image i, patch p
    end for
end for
```

A parser MUST read records in exactly this order. Total record count equals exactly `image_count * sub_images_per_set`.

8 Sub-Image Payload

8.1 Header Dword

The first 4 octets of every sub-image payload form a little-endian 32-bit header dword:

Bits	Field	Description
[0]	palette_mode	See Section 8.1.2
[1]	is_8bit	See Section 8.1.2
[2]	transparency_hint	See Section 8.1.2
[15:3]	width	Native width in pixels (0–8191)
[16]	has_alpha	See Section 8.1.2
[17]	(reserved)	Always 0 in known corpus
[19:18]	row_ptr_width-1	See Section 8.1.3
[31:20]	height	Native height in pixels (0–4095)

Table 4: Logical image header bit layout

8.1.1 Geometry Fields

- **width** (`bits[15:3]`) Native width of this sub-image in pixels. Decoded as: `width = (header_dword >> 3) & 0x1FFF`
- **height** (`bits[31:20]`) Native height of this sub-image in pixels. Decoded as: `height = header_dword >> 20`

8.1.2 Flag Bits

`bit[0]` — `palette_mode`

0: Direct RGB. Colour values are stored in the bitstream as independent R, G, B channel values.

1: Palette mode. Decoded values are indices into an external colour table. See Section 9.8.

`bit[1]` — `is_8bit`

0: 5-bit precision. Channel values are stored as 5-bit integers (0–31) and are multiplied by 8 to produce 8-bit output, giving 32 levels: 0, 8, 16, . . . , 248.

1: 8-bit precision. Channel values are stored as full 8-bit integers (0–255); no scaling is applied.

`bit[2]` — `transparency_hint`

A rendering hint used to select the transparency colour parameter passed to the blitter (`0x00000000` or `0xFFFFFFFF`). This bit does NOT control the alpha decode path and does NOT indicate whether the image has an alpha channel. Decoders producing standalone output (e.g., PNG export) SHOULD ignore this bit.

`bit[16]` — `has_alpha`

0: No alpha channel. The bitstream contains only colour tokens. All output pixels are fully opaque ($\alpha = 255$).

1: Alpha channel present. The bitstream is wrapped by an alpha meta-decoder (Section 9.7) that provides per-pixel alpha values interleaved with colour data.

`bit[17]` — `(reserved)`

Always 0 in the known corpus. Parsers SHOULD ignore this bit.

8.1.3 `row_ptr_width` Field (`bits[19:18]`)

```
row_ptr_width = ((header_dword >> 18) & 0x3) + 1
```

Valid range: 1–4. This is the width in octets of each entry in the per-image row-navigation index that immediately follows the header dword (see Section 9.2).

8.2 Pixel Data

Octets 4 through (`payload_size - 1`) constitute the pixel data bitstream. The first section of this bitstream is the row-navigation index (Section 9.2). The remaining octets hold the RLE channel tokens (Section 9.6) and alpha meta-tokens (Section 9.7).

A decoder MUST treat the pixel data as a bit-level stream read LSB-first within each octet (Section Section 9.3).

9 RLE Pixel Encoding

9.1 Encoding Modes

Three flag bits – `palette_mode` (`bit[0]`), `is_8bit` (`bit[1]`), and `has_alpha` (`bit[16]`) – determine the encoding mode. The following combinations have been observed:

Direct RGB, 5-bit, opaque (`palette_mode`=0, `is_8bit`=0, `has_alpha`=0):

Three colour channels R, G, B encoded per row with 5-bit tokens.

Output: R = ch0, G = ch1, B = ch2, A = 255.

Direct RGB, 5-bit, alpha (`palette_mode`=0, `is_8bit`=0, `has_alpha`=1):

Three colour channels R, G, B with alpha meta-decoder.

Output: R = ch0, G = ch1, B = ch2, A = alpha.

Direct RGB, 8-bit, opaque (`palette_mode`=0, `is_8bit`=1, `has_alpha`=0):

Three colour channels R, G, B encoded per row with 8-bit tokens.

Output: R = ch0, G = ch1, B = ch2, A = 255.

Direct RGB, 8-bit, alpha (`palette_mode`=0, `is_8bit`=1, `has_alpha`=1):

Three colour channels R, G, B with alpha meta-decoder.

Output: R = ch0, G = ch1, B = ch2, A = alpha.

Palette, 8-bit, opaque (`palette_mode`=1, `is_8bit`=1, `has_alpha`=0):

Two channels ch0, ch1 encoded per row with 8-bit tokens.

Output: grayscale from ch1, or palette lookup. See Section 9.8.

Palette, 8-bit, alpha (`palette_mode`=1, `is_8bit`=1, `has_alpha`=1):

Two channels ch0, ch1 with alpha meta-decoder.

Output: grayscale from ch1, or palette lookup + premultiplied alpha. See Section 9.8.

`palette_mode`	`is_8bit`	`has_alpha`	**Channels**	**Alpha**
0	0	0	3 (R,G,B)	none
0	0	1	3 (R,G,B)	meta-decoder
0	1	0	3 (R,G,B)	none
0	1	1	3 (R,G,B)	meta-decoder
1	1	0	2 (ch0,ch1)	none
1	1	1	2 (ch0,ch1)	meta-decoder

Table 5: Channel summary

NOTE: 5-bit palette mode (`palette_mode=1`, `is_8bit=0`) is not present in the known corpus but is handled by the reference implementation as a single channel decoded with 5-bit tokens, palette index = `value >> 3`.

9.2 Row Navigation Index

The pixel data begins at `pixel_ptr` (sub-image payload byte offset +4). It opens with a flat array of `height` entries, each `row_ptr_width` octets wide – the row-navigation index.

Memory Layout:

```
pixel_ptr + 0                           index[0]
pixel_ptr + row_ptr_width               index[1]
...
pixel_ptr + (height-1)*row_ptr_width    index[height-1]
```

Each index entry is an unsigned little-endian integer of `row_ptr_width` octets. The entry encodes the byte offset from its own start position to the start of the corresponding row data.

Row 0 (the topmost display row) is special: its data begins at the position of the LAST index entry:

```
row_data_start[0] = pixel_ptr + (height - 1) * row_ptr_width
```

For subsequent rows ($r \geq 1$):

```
entry_offset = pixel_ptr + (r - 1) * row_ptr_width
row_data_start[r] = entry_offset + index[r-1]
```

In other words, `index[r-1]` is a forward offset from the position of entry (`r-1`) to the start of row `r`'s encoded data.

The last index entry (`index[height-1]`) serves double duty: its byte position is the start of row 0's bitstream, and its value (if read as an integer) is the forward offset to the start of the hypothetical "row after the last" – in practice, the value points past all row data and is not used for navigation.

General formula (0-indexed):

Algorithm 2 Compute start position of row r encoded data

1: **if** $r = 0$ **then**
2: $\quad$ pos $\leftarrow$ pixel_ptr + (height $-$ 1) $\cdot$ row_ptr_width
3: **else**
4: $\quad$ entry_pos $\leftarrow$ pixel_ptr + $(r-1)$ $\cdot$ row_ptr_width
5: $\quad$ pos $\leftarrow$ entry_pos + read_uint(entry_pos, row_ptr_width)
6: **end if**
7: **return** pos

Rows are stored in top-to-bottom display order: row 0 is the topmost scanline. A decoder iterates `r` from `0` to `height-1`.

9.3 Bit-Cursor

The decoder MUST maintain a bit-cursor over the pixel data:

```
struct BitCursor {
    uint8_t  *byte_ptr;   // pointer into the data octet array
    uint8_t   bit_off;    // current bit offset within *byte_ptr (0-7)
};
```

Algorithm 3 Read N bits from the cursor (LSB-first)

1: $value \leftarrow 0$
2: **for** $i \leftarrow 0$ **to** $N - 1$ **do**
3: $\quad value \leftarrow value \mid \Big(((*byte\_ptr \gg bit\_off) \,\&\, 1) \ll i\Big)$
4: $\quad bit\_off \leftarrow bit\_off + 1$
5: $\quad$ **if** $bit\_off = 8$ **then**
6: $\quad\quad byte\_ptr \leftarrow byte\_ptr + 1$
7: $\quad\quad bit\_off \leftarrow 0$
8: $\quad$ **end if**
9: **end for**
10: **return** $value$

This reads bits LSB-first within each octet, assembling a value with the first bit read in the least-significant position.

9.4 Run-Length Bit Width (B)

B is computed once per sub-image from the native width:

Algorithm 4 Compute B from native width

```
clamped ← min(width, 255)
if clamped ≤ 1 then
    return 1
else
    B ← 1
    x ← (clamped − 1) ≫ 1
    while x > 0 do
        B ← B + 1
        x ← x ≫ 1
    end while
    return B
end if
```

This yields

$$B = 1 + \left\lfloor \log_2 \left(\min(width, 255) \right) \right\rfloor \quad \text{for } width \geq 2.$$

Examples:

width	B
1	1
6	3
16	4
128	7
255	8
256	8 (clamped to 255)

Table 6: Examples of B computation

9.5 Channel Interleaving

Each row's encoded data is a SINGLE sequential bitstream shared across all channels. All channels for a row share one `BitCursor`, initialized to `row_data_start[r]`.

The decoder operates pixel-column by pixel-column. For each column position, it checks whether the cumulative decoded count for each channel has reached the current column; if not, it reads new tokens until it has. Channels are decoded in a round-robin fashion:

```
Direct RGB modes:      ch0 (R), ch1 (G), ch2 (B)
Palette 8-bit modes:   ch0, ch1
```

Algorithm 5 Decode one row in 3-channel mode (no alpha)

```
cursor ← BitCursor(row_data_start[r])
bufs ← { [], [], [] }
x ← 0
while x < width do
    for c ∈ {0, 1, 2} do
        while len(bufs[c]) ≤ x do
            decode_token(cursor, bufs[c])
        end while
    end for
    x ← min (len(bufs[0]), len(bufs[1]), len(bufs[2]), width)
end while
```

After the loop, `bufs[0][0..width-1]` contains the R channel, `bufs[1]` the G channel, and `bufs[2]` the B channel. For 2-channel palette mode, replace `[0, 1, 2]` with `[0, 1]`.

IMPORTANT: When the alpha meta-decoder is active (`has_alpha=1`), the alpha meta-decoder wraps the interleaved channel decode. See Section 9.7 for the complete algorithm.

9.6 Token Format

Every token begins with a 2-bit type field, followed by a B-bit `run_count` field, followed by a type-specific value field. All fields are read via the shared `BitCursor` (Section 9.3).

```
type        = read_bits(cursor, 2)
run_count   = read_bits(cursor, B)
pixel_count = run_count + 1
```

The type field is encoded as follows:

type	Mnemonic	Description
0	`FILL`	Solid fill: one value repeated
1	`DELTA_POS`	Delta-positive run
2	`DELTA_NEG`	Delta-negative run
3	`LITERAL`	Literal run: per-pixel values

Table 7: Encoding of the `type` field

The token appends `pixel_count` values to the channel's output buffer. The "prev" value used by delta types is the last value in the channel's output buffer, or 0 if the buffer is empty.

9.6.1 Type 0 – Solid Fill

Algorithm 6 5-bit mode

```
fill5 ← read_bits(cursor, 5)
output_value ← (fill5 · 8) & 0xFF
append pixel_count copies of output_value to the buffer
```

Algorithm 7 8-bit mode

```
fill8 ← read_bits(cursor, 8)
append pixel_count copies of fill8 to the buffer
```

9.6.2 Type 1 – Delta-Positive Run

A "delta run" encodes `pixel_count` successive pixels, each equal to the previous output pixel plus a small positive delta.

Algorithm 8 Decode DELTA token

```
delta_bits ← read_bits(cursor, 3) + 1
if buffer is empty then
    prev ← 0
else
    prev ← last value in the channel buffer
end if
for i ← 0 to pixel_count − 1 do
    raw ← read_bits(cursor, delta_bits)
    if 5-bit mode then
        prev ← (prev + raw · 8) & 0xFF
    else                                          ▷ 8-bit mode
        prev ← (prev + raw) & 0xFF
    end if
    append prev to the buffer
end for
```

9.6.3 Type 2 – Delta-Negative Run

This mode is identical to Type 1 (`DELTA_POS`) except that the decoded delta is *subtracted* from the previous value.

Algorithm 9 Decode DELTA_NEG token

```
delta_bits ← read_bits(cursor, 3) + 1
if buffer is empty then
    prev ← 0
else
    prev ← last value in the channel buffer
end if
for i ← 0 to pixel_count − 1 do
    raw ← read_bits(cursor, delta_bits)
    if 5-bit mode then
        prev ← (prev − raw · 8) & 0xFF
    else                                              ▷ 8-bit mode
        prev ← (prev − raw) & 0xFF
    end if
    append prev to the buffer
end for
```

9.6.4 Type 3 – Literal Run

Each pixel is stored independently.

Algorithm 10 Decode LITERAL token (5-bit mode)

```
for i ← 0 to pixel_count − 1 do
    v5 ← read_bits(cursor, 5)
    append (v5 · 8) & 0xFF to the buffer
end for
```

Algorithm 11 Decode LITERAL token (8-bit mode)

```
for i ← 0 to pixel_count − 1 do
    append read_bits(cursor, 8) to the buffer
end for
```

9.7 Alpha Meta-Decoder

When `has_alpha` (`bit[16]`) is set, the per-row bitstream is not read directly by the channel interleaver. Instead, an alpha meta-decoder sits between the bitstream and the channel decode, providing per-pixel alpha values and controlling when colour tokens are consumed.

The meta-decoder reads a 2-bit `alpha_type` from the cursor before each group of pixels:

alpha_type	Mnemonic	Description
0	`TRANSPARENT`	Transparent run (`alpha=0`, `RGB=0`)
1	`REGULAR`	Explicit alpha via token decoder
2	`OPAQUE`	Opaque run (`alpha=255`)
3	`TRANSPARENT`	Transparent run (same as type 0)

Table 8: Encoding of the alpha_type field

Pseudocode for a full row with 3-channel alpha:

Algorithm 12 Decode one row with alpha channel

```
cursor ← BitCursor(row_data_start[r])
a_buf ← [ ]                                  ▷ alpha output, persistent across segments
r_buf ← [ ]                                                          ▷ final R output
g_buf ← [ ]                                                          ▷ final G output
b_buf ← [ ]                                                          ▷ final B output
ch_bufs ← {[ ], [ ], [ ]}                     ▷ persistent colour channel decode buffers
color_pos ← 0
col ← 0
while col < width do
    alpha_type ← read_bits(cursor, 2)
    if alpha_type = 0 or alpha_type = 3 then
                                                  ▷ Section 9.7.1 — transparent run
        ...
    else if alpha_type = 1 then
                                  ▷ Section Section 9.7.2 — regular (explicit alpha)
        ...
    else
                                                  ▷ Section Section 9.7.3 — opaque run
        ...
    end if
end while
                          ▷ All buffers now have ≥ width entries; truncate to width
```

9.7.1 Alpha Type 0 / 3 – Transparent Run

A B-bit `run_count` is read, giving `pixel_count = run_count + 1` fully transparent pixels:

Algorithm 13 Transparent run (alpha_type = 0 or 3)

```
run_count ← read_bits(cursor, B) + 1
append run_count zeros to a_buf
append run_count zeros to r_buf, g_buf, b_buf
```

Colour channel decode buffers (`ch_bufs`) are NOT consumed. Any previously decoded but unconsumed colour values carry over to the next segment. `color_pos` is NOT advanced.

Algorithm 14 Advance column position

1: `col` $\leftarrow$ `col` + `run_count`

Implementation Warning

Transparent alpha runs do not consume colour tokens. Implementations that advance the colour cursor during transparent segments will desynchronise and produce incorrect output.

9.7.2 Alpha Type 1 – Regular (Explicit Alpha)

A full value token (Section 9.6) is decoded to produce alpha values. The token is decoded directly into `a_buf` (NOT into a temporary buffer), so the prev value correctly carries from previous segments.

Algorithm 15 Compute produced alpha count

1: `alpha_before` $\leftarrow$ `len`(`a_buf`)
2: `decode_token`(`cursor`, `a_buf`) ▷ appends to `a_buf`
3: `count` $\leftarrow$ `len`(`a_buf`) − `alpha_before`

Then the corresponding colour values are decoded via the interleaved channel decoder. The channels MUST be advanced to (`color_pos + count`):

Algorithm 16 Ensure colour channels decoded up to target position

1: `interleave_channels`(`cursor`, `ch_bufs`, `target` = `color_pos` + `count`)

The next `count` colour values from `ch_bufs` are appended to the output:

Algorithm 17 Transfer decoded colour values and advance positions

1: **for** $i \leftarrow 0$ **to** `count` − 1 **do**
2: `r_buf.append`(`ch_bufs`[0][`color_pos` + i])
3: `g_buf.append`(`ch_bufs`[1][`color_pos` + i])
4: `b_buf.append`(`ch_bufs`[2][`color_pos` + i])
5: **end for**
6: `color_pos` $\leftarrow$ `color_pos` + `count`
7: `col` $\leftarrow$ `col` + `count`

9.7.3 Alpha Type 2 – Opaque Run

A B-bit `run_count` is read, giving `pixel_count = run_count + 1` fully opaque pixels:

Algorithm 18 Read run length

```
run_count ← read_bits(cursor, B) + 1
```

Colour values are decoded for this many pixels:

Algorithm 19 Opaque run: emit colour with alpha = 255

```
interleave_channels(cursor, ch_bufs, target = color_pos + run_count)
for i ← 0 to run_count − 1 do
    a_buf.append(0xFF)
    r_buf.append(ch_bufs[0][color_pos + i])
    g_buf.append(ch_bufs[1][color_pos + i])
    b_buf.append(ch_bufs[2][color_pos + i])
end for
color_pos ← color_pos + run_count
col ← col + run_count
```

9.7.4 Alpha Prev Value

In the reference implementation, the alpha token decoder reads the "prev" value from the last byte of the alpha output buffer. This means the prev value carries across alpha segments:

- After a transparent run: `prev = 0`;
- After an opaque run: `prev = 0xFF`;
- After a regular segment: `prev = last decoded alpha value`.

Implementations MUST ensure that `decode_token` for alpha type 1 reads `prev` from the persistent alpha output buffer, NOT from a temporary per-segment buffer. Failure to do this causes compounding delta errors in 8-bit alpha values.

9.7.5 Color Channel Persistence

The colour channel decode buffers (`ch_bufs`) persist across all alpha segments within a row. A token that produces excess colour values carries them into the next segment. Transparent runs do NOT consume from the colour buffers.

The prev value for each colour channel is likewise persistent: it is the last value appended to that channel's buffer, regardless of which alpha segment produced it.

9.8 Palette Post-Processing

When `palette_mode` (`bit[0]`) is set, the decoded channel values are not direct RGB. A post-processing step converts them to RGB output.

The reference implementation appears to dispatch to a routine with four execution paths determined by (a) whether a palette pointer is available, and (b) whether an alpha channel is present.

9.8.1 8-bit Palette Lookup

When a palette table is available and `palette_mode=1` and `is_8bit=1`, two channels (`ch0` and `ch1`) are decoded. The palette index is:

```
index = (ch0 << 8) | ch1
R = palette[index * 3 + 0]
G = palette[index * 3 + 1]
B = palette[index * 3 + 2]
```

The palette is a flat array of RGB triplets (3 bytes per entry).

9.8.2 5-bit Palette Lookup

When `palette_mode=1` and `is_8bit=0`, one channel is decoded. The palette index is derived by dividing the decoded value by 8:

Algorithm 20 Palette lookup from decoded value

1: `index` $\leftarrow$ `decoded_value` $\gg 3$ ▷ 0–31
2: $R \leftarrow$ `palette`[`index` $\cdot\, 3 + 0$]
3: $G \leftarrow$ `palette`[`index` $\cdot\, 3 + 1$]
4: $B \leftarrow$ `palette`[`index` $\cdot\, 3 + 2$]

9.8.3 Grayscale Fallback

When no palette table is available (palette pointer is `NULL`), the reference implementation uses a grayscale fallback:

Algorithm 21 8-bit palette mode

1: $R \leftarrow G \leftarrow B \leftarrow$ `ch1` ▷ the second channel provides the gray value

IMPORTANT: `ch0` MUST still be decoded to keep the bitstream in sync, but its values are discarded in the grayscale fallback.

Algorithm 22 5-bit palette mode

1: $R \leftarrow G \leftarrow B \leftarrow$ `decoded_value`

9.8.4 Premultiplied Alpha

When alpha is present in palette mode, the RGB values from the palette lookup (or grayscale fallback) are premultiplied by the alpha value before output:

Algorithm 23 Apply premultiplied alpha to RGB

if `alpha` $= 255$ **then**
    `out_R` $\leftarrow R$; `out_G` $\leftarrow G$; `out_B` $\leftarrow B$
else if `alpha` $= 0$ **then**
    `out_R` $\leftarrow 0$; `out_G` $\leftarrow 0$; `out_B` $\leftarrow 0$
else
    `out_R` $\leftarrow ((R \cdot \texttt{alpha} + 0x80) \gg 8)$ & $0x\texttt{FF}$
    `out_G` $\leftarrow ((G \cdot \texttt{alpha} + 0x80) \gg 8)$ & $0x\texttt{FF}$
    `out_B` $\leftarrow ((B \cdot \texttt{alpha} + 0x80) \gg 8)$ & $0x\texttt{FF}$
end if

NOTE: the reference implementation seems to use a faster approximation that processes two channels in parallel:

Algorithm 24 Packed ARGB premultiplication step

`ARGB` $\leftarrow (\texttt{alpha} \ll 24) \mid (R \ll 16) \mid (G \ll 8) \mid B$
`br` $\leftarrow (\texttt{ARGB}$ & `0x00FF00FF`$) \cdot \texttt{alpha} +$ `0x00800080`
`g` $\leftarrow (G \mid$ `0x01000000`$) \cdot \texttt{alpha} +$ `0x00800080`
`br` $\leftarrow ((\texttt{br} \gg 8)$ & `0x00FF00FF`$) + (\texttt{br} \gg 8$ & `0x00FF00FF`$)$
`g` $\leftarrow ((\texttt{g} \gg 8)$ & `0x0000FF00`$)$

For a standalone PNG exporter, the simple per-channel formula above is sufficient.

IMPORTANT: Direct RGB modes (`palette_mode=0`) with alpha do NOT apply premultiplied alpha in the decoder. The decoded R, G, B values are the final output values, and the alpha channel is stored separately. Premultiplied alpha is a palette-mode-only post-processing step.

10 DPI Scaling

When `dpi_code != 7` and the draw-flags parameter does NOT have bit 10 (`0x400`) set, the sub-image dimensions MUST be scaled before layout:

Algorithm 25 Compute rendered dimensions from DPI scaling

1: `scale_x` $\leftarrow$ `screen_dpi_x` / `design_dpi`
2: `scale_y` $\leftarrow$ `screen_dpi_y` / `design_dpi`
3: `rendered_width` $\leftarrow \lfloor$ `native_width` $\cdot$ `scale_x` $+ 0.5 \rfloor$
4: `rendered_height` $\leftarrow \lfloor$ `native_height` $\cdot$ `scale_y` $+ 0.5 \rfloor$

Rounding MUST use round-half-up (equivalent to `floor(x + 0.5)`).

Additionally, a cross-platform PNG exporter can ignore DPI scaling entirely and output images at their native resolution.

11 Draw Flags

The `draw_flags` parameter (`uint32`) passed to `RLEDrawToDIB` controls rendering behaviour. Relevant bits:

Bit	Mask	Name	Effect
8	`0x0100`	`ALPHA_DRAW`	Enable alpha/transparent rendering
10	`0x0400`	`NO_DPI_SCALE`	Suppress DPI scaling (Section 10)

Table 9: Control flag bits

A standalone converter does not need draw flags; they control blitter behaviour in the Windows rendering pipeline.

12 Rendering Pipeline

12.1 Single-Image Path

When `sub_images_per_set == 1` (`tile_type == 0x0`):

1. Retrieve sub-image pointer via the record table.
2. Parse header, determine mode.
3. Decode all rows and assemble RGBA output.

12.2 N-Slice Path

When `sub_images_per_set > 1`:

1. Extract fixed-border dimensions (Section 6.3).
2. Split the destination rectangle into up to 9 tile rects.

3. For each tile rect in row-major order, decode and blit the corresponding sub-image.

A standalone converter MAY output each sub-image as a separate PNG file rather than compositing them.

13 Color Sources

The reference renderer recognises three colour-source modes:

type	Meaning
0	No special colour source; pixel data is self-contained.
1	Custom palette: provided by the caller.
2	System colours: a table of RGB triplets built from GetSysColor indices 0–30.

Table 10: Special colour source type

13.1 System-colour table

The system colour table is a flat array of 31 RGB triplets (93 bytes), returned by `GetSysColor`, one entry per index 0-30 in order. Entry `i` occupies bytes `[i*3, i*3+1, i*3+2]` as R, G, B.

A cross-platform implementation MAY substitute equivalent platform colour lookup mechanisms. When no palette is available, the grayscale fallback (Section 9.8.3) produces usable output.

14 Implementation Notes

14.1 Row Navigation

To navigate to row `r` (0-indexed, top-to-bottom):

Algorithm 26 Compute start position of row r (little-endian index)

if $r = 0$ **then**
    pos $\leftarrow$ pixel_ptr + (height $- 1$) $\cdot$ row_ptr_width
else
    entry_pos $\leftarrow$ pixel_ptr + $(r - 1)$ $\cdot$ row_ptr_width
    pos $\leftarrow$ entry_pos + read_uint_le(entry_pos, row_ptr_width)
end if
return pos

14.2 Delta Wrap-Around

Delta operations use 8-bit unsigned wrap-around arithmetic (`(prev +/- delta) & 0xFF`). Implementors MUST NOT clamp delta results.

14.3 Five-Bit Precision

The multiplication by 8 means that 5-bit encoded images can represent only 32 distinct channel levels (`0, 8, 16, ..., 248`).

14.4 Alpha Buffer Persistence

The alpha output buffer and colour channel decode buffers MUST persist for the entire row. Creating fresh buffers per alpha segment causes incorrect `prev` values.

14.5 8-bit Palette Channel Order

In 8-bit palette mode, the grayscale value comes from `ch1` (the SECOND decoded channel), not `ch0`. `ch0` MUST still be decoded to keep the bit cursor synchronized.

14.6 Channel-to-Colour Mapping for Direct RGB

The three decoded channels map to: `ch0` = Red, `ch1` = Green, `ch2` = Blue. For PNG output in RGBA format, this maps directly: R=`ch0`, G=`ch1`, B=`ch2`. (The reference implementation writes to a BGRX-32 DIB where the DWORD `0xFF_ch0_ch1_ch2` places `ch0` at the R byte position and `ch2` at the B byte position in the BGRA memory layout.)

14.7 Token Overflow

A decoded token may produce more pixels than the remaining columns in the row. A decoder MUST clip each channel buffer to exactly `width` pixels and MUST NOT fail.

14.8 Minimal PNG Exporter Recipe

1. Read file header; verify magic; parse `image_count`, `size_field_width`, `tile_type`.
2. For each of the (`image_count * sub_images_per_set`) records:
   2.1 Read `payload_size` (`size_field_width` octets, LE).
   2.2 Read `payload_size` octets as the sub-image payload.
3. For each sub-image payload:

3.1 Parse the 4-octet header dword; extract `width`, `height`, `flags`, `row_ptr_width` (Section 8.1).

3.2 Determine encoding mode (Section 9.1).

3.3 Compute B (Section 9.4).

3.4 Navigate to each row `r = 0 .. height-1` (Section 9.2).

3.5 For each row, decode channels (Section 9.5, Section 9.6). If `has_alpha`, use the alpha meta-decoder (Section 9.7). If `palette_mode`, apply post-processing (Section 9.8).

3.6 Write the RGBA row to the output image.

15 Security Considerations

- Input validation: A parser MUST verify the magic bytes (Section 5.1) before processing any other field. A parser MUST NOT access pixel data beyond the bounds indicated by `payload_size`.
- Integer overflow: `width` (max 8191) and `height` (max 4095) are each bounded by 13 and 12 bits respectively; their product (max $\approx$ 33.5 million) fits in a 32-bit unsigned integer. Implementations MUST check that computed buffer sizes do not exceed available memory before allocation.
- Bitstream over-read: Because tokens may straddle octet boundaries, a decoder MUST ensure the `BitCursor` does not advance past `pixel_ptr + payload_size - 4`. Reads beyond this boundary produce undefined results.
- Malformed run counts: A run of `run_count + 1` pixels where $run\_count = 2^B - 1$ could produce up to `min(256, width)` pixels per token. The decoder MUST track the cumulative pixel count per channel and MUST NOT write beyond the end of the row buffer.
- Row-navigation index bounds: Each index entry value, when added to the entry's own address, MUST yield an address within the bounds of the sub-image payload.

16 Appendices

A Annotated Hex Dump (`rle_4000_287_1033.bin`)

A small single-tile corpus file (76 bytes). Renders as a 7×7 diamond-like shape in a single colour (R=40, G=72, B=136) on a transparent background. Mode: direct RGB, 5-bit precision, alpha.

File header and record framing

Offset	Hex	Annotation
0000	52 4C 45	Magic “RLE”
0003	01	image_count = 1
0004	00	size_field_width=1, tile_type=0x0 (1×1)
0005	00	dpi_code=0 (96 DPI)

Record 0

Offset	Hex	Annotation
0006	45	payload_size = 69

Sub-image payload (69 bytes, offset 0x07–0x4B)

Offset	Hex	Annotation
0007	38 00 71 00	Header dword = 0x00710038

Header decode (0x00710038 in binary, LSB first):

Field	Value	Meaning
bit[0]	0	palette_mode=0 (direct RGB)
bit[1]	0	is_8bit=0 (5-bit precision)
bit[2]	0	transparency_hint=0
bits[15:3]	7	width=7 pixels
bit[16]	1	has_alpha=1
bit[17]	0	(reserved)
bits[19:18]	0	row_ptr_width=1
bits[31:20]	7	height=7 pixels

$B = 3$ (for width=7: $\lfloor \log_2(6) \rfloor + 1 = 3$).

Row navigation index (7 entries × 1 byte, offset 0x0B–0x11)

Offset	Hex	Annotation
000B	10	index[0] = 16
000C	19	index[1] = 25
000D	1E	index[2] = 30
000E	23	index[3] = 35
000F	28	index[4] = 40
0010	31	index[5] = 49
0011	86	index[6] = 134

Row data positions (Section 9.2):

```
Row 0: pixel_ptr + 6*1 = offset 0x11      (last index entry)
Row 1: 0x0B + 16       = offset 0x1B
Row 2: 0x0C + 25       = offset 0x25
Row 3: 0x0D + 30       = offset 0x2B
Row 4: 0x0E + 35       = offset 0x31
Row 5: 0x0F + 40       = offset 0x37
Row 6: 0x10 + 49       = offset 0x41
```

Encoded pixel data (offset 0x11–0x4B)

```
0011    86 14 92 48 44 86 14 92  48 04 0A 15 94 50 04 0A
0021    15 94 50 04 40 C2 02 13  8C 00 44 A1 82 12 8A 04
0031    40 C2 02 13 8C 00 0A 15  94 50 04 0A 15 94 50 04
0041    86 14 92 48 44 86 14 92  48 04 FE
```

B Worked Decode Example

This example traces the decode of Row 0 of `rle_4000_287_1033.bin` (Appendix A). Mode: 5-bit, direct RGB, `has_alpha`=1, $B = 3$, `width`=7.

Row 0 bitstream starts at offset `0x11`. Raw bytes: `86 14 92 48 44`.

Bit order

The bit-cursor reads LSB-first within each byte:

Byte	Bits read (LSB-first)
`0x86`	0 1 1 0 0 0 0 1
`0x14`	0 0 1 0 1 0 0 0
`0x92`	0 1 0 0 1 0 0 1
`0x48`	0 0 0 1 0 0 1 0
`0x44`	0 0 1 0 0 0 1 0

(Note: bit 0 is the rightmost bit in the hex representation, but the bits are shown left-to-right here in the order they are read.)

Step-by-step decode (Row 0)

Step 1 (bit 0). Read `alpha_type` (2 bits): `10b` = 2 (`OPAQUE`).
Read `run_count` (3 bits): `001b` = 1 ⇒ `pixel_count` = 2.

Step 2 (bit 5). Decode 2 pixels of colour via interleaved tokens.

`ch0` (R) token: `type`=2 bits → `00b`=0 (`FILL`), `run`=3 bits → `001b`=1 (`pixel_count`=2), `val5`=5 bits → `00101b`=5 ⇒ value=40.
`R = [40, 40]`

`ch1` (G) token: `type`=`00b`=0 (`FILL`), `run`=`001b`=1 (`pixel_count`=2), `val5`=`01001b`=9 ⇒ value=72.
`G = [72, 72]`

`ch2` (B) token: `type`=`00b`=0 (`FILL`), `run`=`001b`=1 (`pixel_count`=2), `val5`=`10001b`=17 ⇒ value=136.
`B = [136, 136]`

Output pixels 0–1: R=40, G=72, B=136, A=255 (opaque).

Step 3 (bit 35). Read `alpha_type` (2 bits): `00b` = 0 (`TRANSPARENT`).
Read `run_count` (3 bits): `010b` = 2 ⇒ `pixel_count` = 3.
Output pixels 2–4: R=0, G=0, B=0, A=0 (transparent).
`color_pos` stays at 2 (no colour tokens consumed).

Step 4 (bit 40). Read `alpha_type` (2 bits): `10b` = 2 (`OPAQUE`).
Read `run_count` (3 bits): `001b` = 1 ⇒ `pixel_count` = 2.

`ch0` (R) token: `FILL`, `run`=1, `val5`=5 ⇒ value=40. `R = [40, 40]`
`ch1` (G) token: `FILL`, `run`=1, `val5`=9 ⇒ value=72. `G = [72, 72]`
`ch2` (B) token: `FILL`, `run`=1, `val5`=17 ⇒ value=136. `B = [136, 136]`

Output pixels 5–6: R=40, G=72, B=136, A=255 (opaque).

`col` = 7 = `width`, row complete. 75 bits consumed (9 bytes + 3 bits).

Final Row 0

Pixel	R	G	B	A
0	40	72	136	255
1	40	72	136	255
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	40	72	136	255
6	40	72	136	255

Rendered 7×7 image

The full 7×7 image (`.` = transparent, `#` = opaque 284888):

```
# # . . . # #
# # # . # # #
. # # # # # .
. . # # # . .
. # # # # # .
# # # . # # #
# # . . . # #
```

C Reference Python Implementation

The following is a complete, dependency-free Python 3 implementation (requires only the standard library: `struct`, `zlib`, `os`, `sys`, `argparse`). It has been verified to produce pixel-perfect output for all 380 single-tile files in the corpus (380/380 exact matches against oracle output).

Usage:

```
python3 rle_to_png.py <input.bin> [-o output_dir] [-p palette]
```

```
#!/usr/bin/env python3
"""
rle_to_png.py  --  Convert RLE-DIB files to PNG images.

Decodes the proprietary "RLE" image format used for Windows
themed-control artwork.  Despite the name, the encoding is NOT
standard RLE -- it uses a custom bitstream with solid fills,
signed deltas, and per-pixel literals across independently
interleaved channels.

Encoding variants:
  5-bit mode (header bit1=0): values are 5-bit * 8, range 0-248
  8-bit mode (header bit1=1): values are full 8-bit, range 0-255

Channel configurations:
  3-channel direct RGB  (bit0=0): channels decoded as R, G, B
  1-channel palette     (bit0=1, 5-bit): palette index = value >> 3
  2-channel palette     (bit0=1, 8-bit): palette index
                                         = (ch0 << 8) | ch1

Alpha channel (bit16=1) is handled by a meta-decoder above the
value tokens.
"""

import sys
import os
import struct
import zlib
import argparse


# --- Minimal pure-Python PNG writer (no external dependencies) ---

def _png_chunk(tag: bytes, data: bytes) -> bytes:
    length = struct.pack('>I', len(data))
    crc = struct.pack('>I', zlib.crc32(tag + data) & 0xFFFFFFFF)
    return length + tag + data + crc


def write_png(path, width, height, rgba_rows):
    """Write an RGBA PNG file."""
    signature = b'\x89PNG\r\n\x1a\n'
    ihdr_data = (struct.pack('>II', width, height)
                 + bytes([8, 6, 0, 0, 0]))
```

```
    ihdr = _png_chunk(b'IHDR', ihdr_data)
    raw = b''.join(b'\x00' + row for row in rgba_rows)
    compressed = zlib.compress(raw, 9)
    idat = _png_chunk(b'IDAT', compressed)
    iend = _png_chunk(b'IEND', b'')
    with open(path, 'wb') as f:
        f.write(signature + ihdr + idat + iend)


# --- Bit-cursor (reads LSB-first within each octet) ---

class BitCursor:
    __slots__ = ('data', 'byte_pos', 'bit_off')

    def __init__(self, data, byte_pos):
        self.data = data
        self.byte_pos = byte_pos
        self.bit_off = 0

    def read_bits(self, n):
        value = 0
        for i in range(n):
            if self.byte_pos < len(self.data):
                bit = (self.data[self.byte_pos]
                       >> self.bit_off) & 1
            else:
                bit = 0
            value |= bit << i
            self.bit_off += 1
            if self.bit_off == 8:
                self.byte_pos += 1
                self.bit_off = 0
        return value


# --- Run-length bit-width B ---

def compute_B(width):
    clamped = min(width, 255)
    if clamped <= 1:
        return 1
    B = 1
    x = (clamped - 1) >> 1
    while x > 0:
        B += 1
        x >>= 1
    return B
```

```
# --- Token-level decoder ---

def _decode_token(cursor, buf, B, is_8bit):
    prev = buf[-1] if buf else 0
    token_type = cursor.read_bits(2)
    run_count = cursor.read_bits(B)
    pixel_count = run_count + 1

    if token_type == 0:          # SOLID FILL
        if is_8bit:
            val = cursor.read_bits(8)
        else:
            val = (cursor.read_bits(5) * 8) & 0xFF
        buf.extend([val] * pixel_count)

    elif token_type == 1:        # DELTA POSITIVE
        db = cursor.read_bits(3) + 1
        for _ in range(pixel_count):
            raw = cursor.read_bits(db)
            if is_8bit:
                prev = (prev + raw) & 0xFF
            else:
                prev = (prev + raw * 8) & 0xFF
            buf.append(prev)

    elif token_type == 2:        # DELTA NEGATIVE
        db = cursor.read_bits(3) + 1
        for _ in range(pixel_count):
            raw = cursor.read_bits(db)
            if is_8bit:
                prev = (prev - raw) & 0xFF
            else:
                prev = (prev - raw * 8) & 0xFF
            buf.append(prev)

    else:                        # LITERAL
        for _ in range(pixel_count):
            if is_8bit:
                val = cursor.read_bits(8)
            else:
                val = (cursor.read_bits(5) * 8) & 0xFF
            buf.append(val)
            prev = val
```

```
# --- N-channel interleaver ---

def _interleave_nch(cursor, ch_bufs, target, B, is_8bit):
    n_ch = len(ch_bufs)
    x = min(len(b) for b in ch_bufs)
    while x < target:
        for c in range(n_ch):
            while len(ch_bufs[c]) <= x:
                _decode_token(cursor, ch_bufs[c], B, is_8bit)
        x = min(len(b) for b in ch_bufs)
        x = min(x, target)


# --- Per-row decoders (no alpha) ---

def _decode_row_3ch(cursor, width, B, is_8bit):
    bufs = [[], [], []]
    x = 0
    while x < width:
        for c in range(3):
            while len(bufs[c]) <= x:
                _decode_token(cursor, bufs[c], B, is_8bit)
        x = min(len(bufs[0]), len(bufs[1]),
                len(bufs[2]))
        x = min(x, width)
    return bufs[0][:width], bufs[1][:width], bufs[2][:width]


def _decode_row_1ch(cursor, width, B, is_8bit):
    buf = []
    while len(buf) < width:
        _decode_token(cursor, buf, B, is_8bit)
    return buf[:width]


def _decode_row_2ch(cursor, width, B):
    bufs = [[], []]
    x = 0
    while x < width:
        for c in range(2):
            while len(bufs[c]) <= x:
                _decode_token(cursor, bufs[c], B,
                              is_8bit=True)
        x = min(len(bufs[0]), len(bufs[1]))
        x = min(x, width)
    return bufs[0][:width], bufs[1][:width]
```

```
# --- Per-row decoders (with alpha meta-decoder) ---

def _decode_row_3ch_alpha(cursor, width, B, is_8bit):
    r_buf, g_buf, b_buf, a_buf = [], [], [], []
    ch_bufs = [[], [], []]
    color_pos = 0
    col = 0
    while col < width:
        alpha_type = cursor.read_bits(2)
        if alpha_type == 0 or alpha_type == 3:
            run = cursor.read_bits(B) + 1
            n = min(run, width - col)
            a_buf.extend([0] * n)
            r_buf.extend([0] * n)
            g_buf.extend([0] * n)
            b_buf.extend([0] * n)
            col += run
        elif alpha_type == 1:
            ab = len(a_buf)
            _decode_token(cursor, a_buf, B, is_8bit)
            count = len(a_buf) - ab
            target = color_pos + count
            _interleave_nch(cursor, ch_bufs, target,
                            B, is_8bit)
            n = min(count, width - col)
            for i in range(n):
                r_buf.append(ch_bufs[0][color_pos + i])
                g_buf.append(ch_bufs[1][color_pos + i])
                b_buf.append(ch_bufs[2][color_pos + i])
            color_pos += count
            col += count
        else:  # alpha_type == 2
            run = cursor.read_bits(B) + 1
            target = color_pos + run
            _interleave_nch(cursor, ch_bufs, target,
                            B, is_8bit)
            n = min(run, width - col)
            for i in range(n):
                a_buf.append(0xFF)
                r_buf.append(ch_bufs[0][color_pos + i])
                g_buf.append(ch_bufs[1][color_pos + i])
                b_buf.append(ch_bufs[2][color_pos + i])
            color_pos += run
            col += run
    return (r_buf[:width], g_buf[:width],
            b_buf[:width], a_buf[:width])
```

```
def _decode_row_1ch_alpha(cursor, width, B, is_8bit):
    val_buf, a_buf = [], []
    ch_buf = []
    color_pos = 0
    col = 0
    while col < width:
        alpha_type = cursor.read_bits(2)
        if alpha_type == 0 or alpha_type == 3:
            run = cursor.read_bits(B) + 1
            n = min(run, width - col)
            a_buf.extend([0] * n)
            val_buf.extend([0] * n)
            col += run
        elif alpha_type == 1:
            ab = len(a_buf)
            _decode_token(cursor, a_buf, B, is_8bit)
            count = len(a_buf) - ab
            while len(ch_buf) < color_pos + count:
                _decode_token(cursor, ch_buf, B, is_8bit)
            n = min(count, width - col)
            for i in range(n):
                val_buf.append(ch_buf[color_pos + i])
            color_pos += count
            col += count
        else:
            run = cursor.read_bits(B) + 1
            while len(ch_buf) < color_pos + run:
                _decode_token(cursor, ch_buf, B, is_8bit)
            n = min(run, width - col)
            for i in range(n):
                a_buf.append(0xFF)
                val_buf.append(ch_buf[color_pos + i])
            color_pos += run
            col += run
    return val_buf[:width], a_buf[:width]


def _decode_row_2ch_alpha(cursor, width, B):
    ch0_buf, ch1_buf, a_buf = [], [], []
    ch_bufs = [[], []]
    color_pos = 0
    col = 0
    while col < width:
        alpha_type = cursor.read_bits(2)
        if alpha_type == 0 or alpha_type == 3:
```

```
            run = cursor.read_bits(B) + 1
            n = min(run, width - col)
            a_buf.extend([0] * n)
            ch0_buf.extend([0] * n)
            ch1_buf.extend([0] * n)
            col += run
        elif alpha_type == 1:
            ab = len(a_buf)
            _decode_token(cursor, a_buf, B, is_8bit=True)
            count = len(a_buf) - ab
            target = color_pos + count
            _interleave_nch(cursor, ch_bufs, target,
                            B, is_8bit=True)
            n = min(count, width - col)
            for i in range(n):
                ch0_buf.append(
                    ch_bufs[0][color_pos + i])
                ch1_buf.append(
                    ch_bufs[1][color_pos + i])
            color_pos += count
            col += count
        else:
            run = cursor.read_bits(B) + 1
            target = color_pos + run
            _interleave_nch(cursor, ch_bufs, target,
                            B, is_8bit=True)
            n = min(run, width - col)
            for i in range(n):
                a_buf.append(0xFF)
                ch0_buf.append(
                    ch_bufs[0][color_pos + i])
                ch1_buf.append(
                    ch_bufs[1][color_pos + i])
            color_pos += run
            col += run
    return ch0_buf[:width], ch1_buf[:width], a_buf[:width]


# --- Palette / color conversion ---

def _premultiply(c, a):
    return ((c * a + 0x80) >> 8) & 0xFF


# --- Decode one sub-image payload ---

def decode_subimage(payload, palette=None):
```

```
    if len(payload) < 4:
        raise ValueError('Sub-image payload too short')

    hdr = struct.unpack_from('<I', payload, 0)[0]

    palette_mode  = (hdr >> 0) & 1
    is_8bit       = (hdr >> 1) & 1
    has_alpha     = (hdr >> 16) & 1
    width         = (hdr >> 3) & 0x1FFF
    row_ptr_width = ((hdr >> 18) & 0x3) + 1
    height        = hdr >> 20

    if width == 0 or height == 0:
        return (width, height, [])

    B = compute_B(width)
    index_start = 4

    def row_cursor(r):
        if r == 0:
            pos = (index_start
                   + (height - 1) * row_ptr_width)
        else:
            entry_off = (index_start
                         + (r - 1) * row_ptr_width)
            entry_val = int.from_bytes(
                payload[entry_off:entry_off
                        + row_ptr_width], 'little')
            pos = entry_off + entry_val
        return BitCursor(payload, pos)

    rgba_rows = []
    for r in range(height):
        cursor = row_cursor(r)
        row_bytes = bytearray(width * 4)

        if palette_mode == 0:
            if has_alpha:
                rc, gc, bc, ac = \
                    _decode_row_3ch_alpha(
                        cursor, width, B,
                        bool(is_8bit))
                for x in range(width):
                    row_bytes[x*4]   = rc[x]
                    row_bytes[x*4+1] = gc[x]
                    row_bytes[x*4+2] = bc[x]
                    row_bytes[x*4+3] = ac[x]
```

```
            else:
                rc, gc, bc = _decode_row_3ch(
                    cursor, width, B, bool(is_8bit))
                for x in range(width):
                    row_bytes[x*4]   = rc[x]
                    row_bytes[x*4+1] = gc[x]
                    row_bytes[x*4+2] = bc[x]
                    row_bytes[x*4+3] = 255
        else:
            if is_8bit:
                if has_alpha:
                    c0, c1, ac = \
                        _decode_row_2ch_alpha(
                            cursor, width, B)
                else:
                    c0, c1 = _decode_row_2ch(
                        cursor, width, B)
                    ac = None
                for x in range(width):
                    if palette is not None:
                        idx = (c0[x] << 8 | c1[x]) * 3
                        pr = (palette[idx]
                              if idx < len(palette)
                              else 0)
                        pg = (palette[idx + 1]
                              if idx+1 < len(palette)
                              else 0)
                        pb = (palette[idx + 2]
                              if idx+2 < len(palette)
                              else 0)
                    else:
                        pr = pg = pb = c1[x]
                    if ac is not None:
                        a = ac[x]
                        if a == 0:
                            pr = pg = pb = 0
                        elif a != 0xFF:
                            pr = _premultiply(pr, a)
                            pg = _premultiply(pg, a)
                            pb = _premultiply(pb, a)
                        row_bytes[x*4+3] = a
                    else:
                        row_bytes[x*4+3] = 255
                    row_bytes[x*4]   = pr
                    row_bytes[x*4+1] = pg
                    row_bytes[x*4+2] = pb
            else:
```

```
                if has_alpha:
                    vc, ac = _decode_row_1ch_alpha(
                        cursor, width, B, False)
                else:
                    vc = _decode_row_1ch(
                        cursor, width, B, False)
                    ac = None
                for x in range(width):
                    if palette is not None:
                        idx = (vc[x] >> 3) * 3
                        pr = (palette[idx]
                              if idx < len(palette)
                              else 0)
                        pg = (palette[idx + 1]
                              if idx+1 < len(palette)
                              else 0)
                        pb = (palette[idx + 2]
                              if idx+2 < len(palette)
                              else 0)
                    else:
                        pr = pg = pb = vc[x]
                    if ac is not None:
                        a = ac[x]
                        if a == 0:
                            pr = pg = pb = 0
                        elif a != 0xFF:
                            pr = _premultiply(pr, a)
                            pg = _premultiply(pg, a)
                            pb = _premultiply(pb, a)
                        row_bytes[x*4+3] = a
                    else:
                        row_bytes[x*4+3] = 255
                    row_bytes[x*4]   = pr
                    row_bytes[x*4+1] = pg
                    row_bytes[x*4+2] = pb

            rgba_rows.append(bytes(row_bytes))

        return (width, height, rgba_rows)


    # --- Parse the file header and image records ---

    TILE_COUNT_TABLE = [
        1, 2, 2, 4, 2, 3, 4, 6, 2, 4, 3, 6, 4, 6, 6, 9
    ]
```

```
def parse_rle_file(data):
    if len(data) < 6:
        raise ValueError('File too short')
    if data[0:3] != b'RLE':
        raise ValueError(
            f'Bad magic: {data[0:3]!r}')

    image_count      = data[3] & 0x7F
    size_field_width = (data[4] & 0x3) + 1
    tile_type        = (data[4] >> 2) & 0xF
    tile_count       = TILE_COUNT_TABLE[tile_type]

    pos = 6
    records = []
    for i in range(image_count):
        for p in range(tile_count):
            if pos + size_field_width > len(data):
                raise ValueError('Truncated')
            payload_size = int.from_bytes(
                data[pos:pos + size_field_width],
                'little')
            pos += size_field_width
            if pos + payload_size > len(data):
                raise ValueError('Truncated')
            records.append(
                (i, p, data[pos:pos + payload_size]))
            pos += payload_size
    return records


# --- Main entry point ---

def convert(input_path, output_dir, palette=None):
    with open(input_path, 'rb') as f:
        data = f.read()
    records = parse_rle_file(data)
    stem = os.path.splitext(
        os.path.basename(input_path))[0]
    os.makedirs(output_dir, exist_ok=True)
    written = []
    for img_idx, patch_idx, payload in records:
        try:
            w, h, rgba = decode_subimage(
                payload, palette)
        except Exception as e:
            print(f'  Warning: {e}',
```

```
                  file=sys.stderr)
            continue
        if w == 0 or h == 0:
            continue
        if len(records) == 1:
            name = f'{stem}.png'
        else:
            name = (f'{stem}_img{img_idx}'
                    f'_patch{patch_idx}.png')
        out = os.path.join(output_dir, name)
        write_png(out, w, h, rgba)
        written.append(out)
        print(f'  Wrote {out}  ({w}x{h})')
    return written


def main():
    parser = argparse.ArgumentParser(
        description='Convert RLE-DIB files to PNG.')
    parser.add_argument(
        'input', nargs='+',
        help='RLE-DIB .bin file(s) to convert')
    parser.add_argument(
        '-o', '--output-dir', default=None,
        help='Output directory for PNG files')
    parser.add_argument(
        '-p', '--palette', default=None,
        help='Path to syscolors.bin palette file')
    args = parser.parse_args()

    palette = None
    if args.palette:
        with open(args.palette, 'rb') as f:
            palette = f.read()

    for path in args.input:
        out_dir = (args.output_dir
                   or os.path.dirname(
                       os.path.abspath(path)))
        print(f'Converting {path} ...')
        try:
            written = convert(
                path, out_dir, palette)
            if not written:
                print('  (no images produced)')
        except Exception as e:
            print(f'  Error: {e}',
```

```
                    file=sys.stderr)


if __name__ == '__main__':
    main()
```