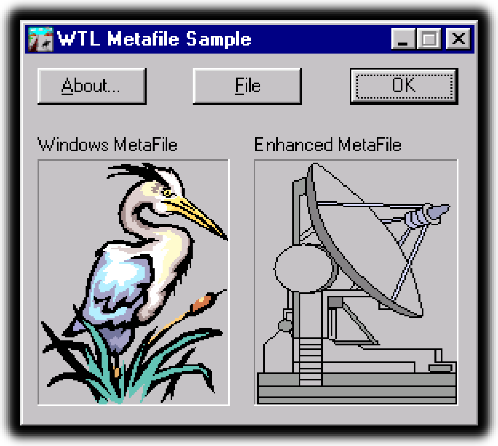
# CHAPTER 18: METAFILES

## Metafiles vs Bitmaps

Metafiles are like vector graphics, while bitmaps are like raster graphics. Metafiles are built from a series of binary records corresponding to graphics function calls, whereas bitmaps are made of pixels, usually from real-world images. Essentially, metafiles describe **how to draw**, and bitmaps describe **what is already drawn**.

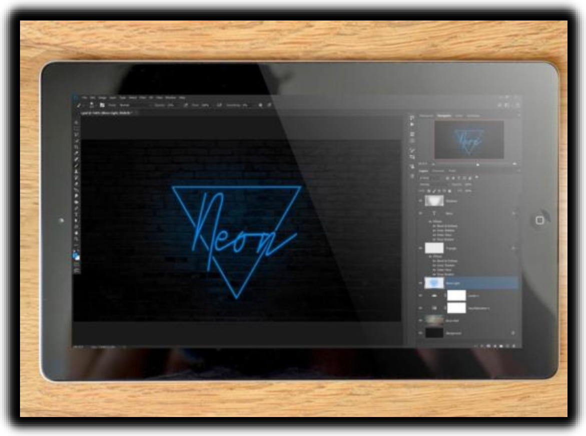


## I. Creating and Editing

Paint programs create bitmaps, while drawing programs create metafiles.

In a drawing program, you can easily move or edit individual graphical objects because they are stored as separate records.

In a paint program, you are generally limited to moving or removing rectangular sections of the bitmap.



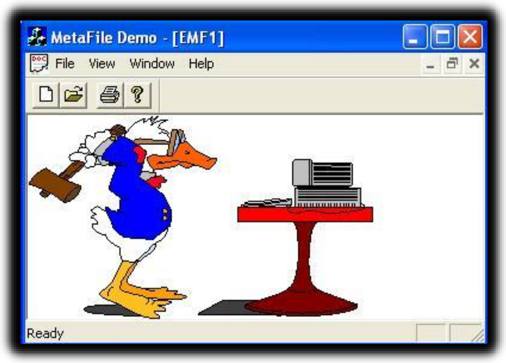
## II. Scaling

Metafiles can be scaled to any size without losing resolution because they are based on drawing commands.

Bitmaps cannot be scaled without losing quality, as enlarging them stretches the pixels.

## III. Converting

You can convert a metafile into a bitmap, but some detail may be lost. Converting a bitmap into a metafile is much harder and usually requires a lot of processing.



## IV. Uses

Metafiles are commonly used for sharing images between programs through the clipboard. They can also exist on disk as clip art.

Metafiles are smaller in size and more device-independent than bitmaps, making them the preferred format for sharing and storing images.

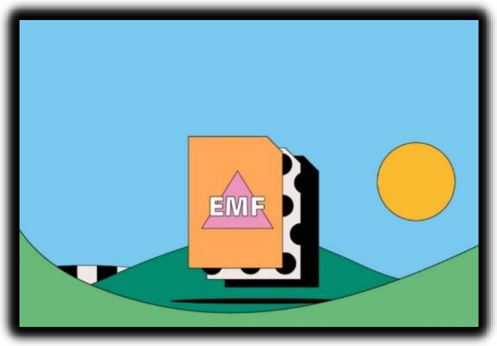
## V. Windows Metafile Formats

Windows supports two metafile formats.

The original metafile has been available since Windows 1.0.

The **enhanced metafile** was developed for 32-bit Windows and includes several improvements over the old format.

It is recommended to use the enhanced metafile whenever possible.



# UNDERSTANDING THE OLD METAFILE FORMAT

## Metafiles: Blueprints for Vector Graphics

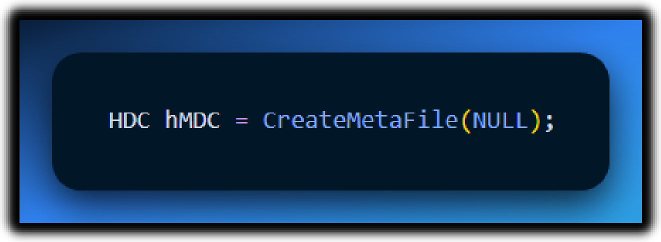
A metafile stores **drawing commands**, not pixel data, acting as a blueprint for recreating an image.

**Advantages over bitmaps:**

- **Device Independence:** Looks the same on screens, printers, and other devices.
- **Compactness:** Less storage needed, especially for simple graphics.
- **Scalability:** Can resize without losing quality since commands are recalculated.

## Working with Memory Metafiles

**I. Create a Metafile Device Context (MDC):**



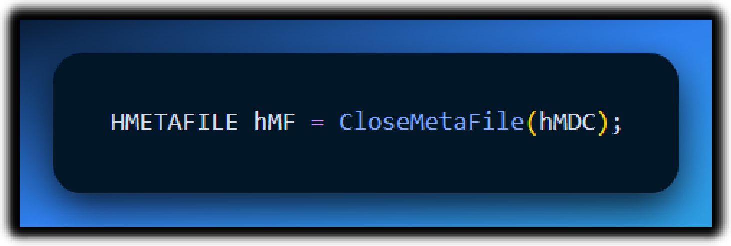
```
HDC hMDC = CreateMetaFile(NULL);
```

Provides a virtual canvas to record drawing commands.

**II. Draw Using GDI Functions:**

Functions like LineTo, Rectangle, TextOut, etc., are recorded, **not displayed immediately**.

**III. Close the MDC:**



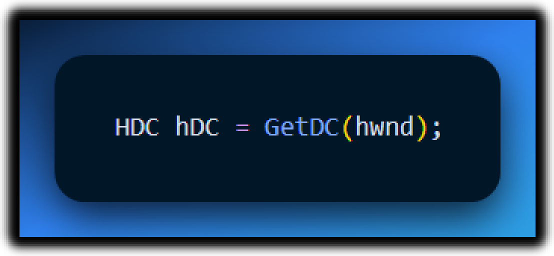
```
HMETAFILE hMF = CloseMetaFile(hMDC);
```

Finalizes the metafile and returns a handle for later use.

---

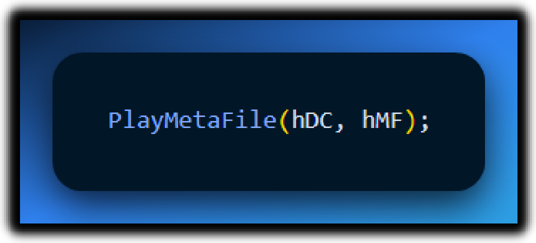
## Rendering (Playing) a Metafile

**I. Get a Real Device Context:**



```
HDC hDC = GetDC(hwnd);
```

**II. Play the Metafile:**



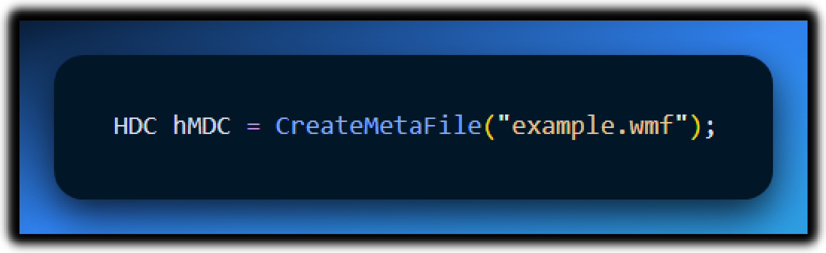
```
PlayMetaFile(hDC, hMF);
```

Executes the recorded GDI commands, rendering the image on the target device.

---

## Saving as a Disk Metafile

CreateMetaFile accepts a **filename** to save the metafile on disk:



```
HDC hMDC = CreateMetaFile("example.wmf");
```

- Windows handles the file I/O automatically.
- If NULL is passed, the metafile stays in memory.

---

## Key Points to Remember

- Metafiles **record commands**, not pixels.
- **Memory metafiles** are temporary; **disk metafiles** persist for later use.
- The **old WMF format** works, but **Enhanced Metafile (EMF)** is preferred for modern applications.

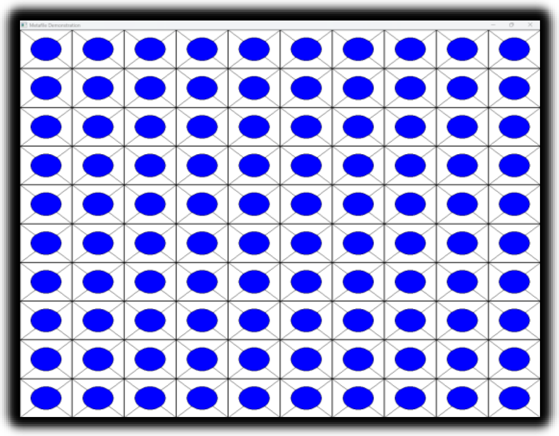
## METAFILE.C – Essentials 🎨

### I. Core Idea

- **Metafiles** store **drawing commands**, not pixels, allowing you to replay graphics multiple times (like a "macro" for GDI drawing).
- METAFILE.C demonstrates **in-memory metafiles**, then shows how to **switch to disk-based metafiles** for large drawings.

### II. Program Flow

1. **Create Metafile**
   - ✓ Use CreateMetaFile(NULL) for memory-based, or CreateMetaFile("example.wmf") for disk-based.
   - ✓ Record GDI commands: rectangles, lines, ellipses, fills, etc.
2. **Playback**
   - ✓ WM_PAINT triggers PlayMetaFile to draw the recorded commands on the window.
   - ✓ Anisotropic mapping mode allows flexible scaling and viewport shifts.
   - ✓ Can tile or reuse the same metafile multiple times efficiently.
3. **Cleanup**
   - ✓ CloseMetaFile finalizes the metafile.
   - ✓ DeleteMetaFile deletes the handle (memory metafiles) without touching disk files.
   - ✓ DeleteFile removes the disk file if desired.



## III. Advanced Concepts

- **Disk vs Memory Metafile**: Disk reduces RAM use but needs file access each time.
- **METAFILEPICT**: Wraps an old metafile with mapping mode and dimensions for clipboard operations. Provides:
  - ✓ Scaling information
  - ✓ Mapping flexibility
  - ✓ Clipboard compatibility
- **Clipboard Tips**:
  - ✓ Copy: Put METAFILEPICT structure on clipboard.
  - ✓ Paste: Retrieve, adjust mapping if needed, play with PlayMetaFile.

## IV. Challenges & Notes

- **Old WMF format** has limitations:
  - ✓ Hard to determine size from handle alone
  - ✓ Limited clipboard flexibility
- **Enhanced Metafile (EMF)** is preferred for modern apps (more features, better scaling, broader compatibility).
- **Performance**: For very large graphics, consider alternative formats (bitmaps) for speed.

## V. Takeaways for WinAPI Coding

- **Think in commands, not pixels.** Metafiles record actions, not final results.
- **Resource management is key.** Always close and delete metafiles, fonts, brushes, and pens to avoid leaks.
- **Use mapping modes wisely.** Anisotropic mapping lets you scale and tile, but you must manage coordinate systems.
- **Experiment safely.** Disk vs memory, clipboard, and tiling are all ways to explore how WinAPI handles graphics efficiently.

# METAFILES AND THE CLIPBOARD

This chapter deals with a complex problem: How do you copy a drawing from one program (like Excel) and paste it into another (like Word) while keeping the size correct?

If you just copy pixels (Bitmap), the image gets blurry when resized.

If you copy commands (Metafile), the image stays sharp, but you need to tell the other program how big the drawing is supposed to be.

## I. The Structure: METAFILEPICT

A raw Metafile is just a list of commands (Draw Line, Draw Circle). It has no concept of "Physical Size." To fix this, Windows wraps the metafile in a structure called **METAFILEPICT** before putting it on the clipboard.

It contains 4 fields:

1. **mm:** The Mapping Mode (e.g., Millimeters, Inches, Pixels).
2. **xExt:** The Width of the image.
3. **yExt:** The Height of the image.
4. **hMF:** The Handle to the Metafile itself.

## II. Creating the Metafile (The Copy Side)

When your program wants to copy a drawing to the clipboard:

**Step A: Create the Metafile**

1. Call CreateMetaFile(NULL) to create a metafile in memory (not on disk).
2. Set the Window Origin (SetWindowOrgEx) and Extent (SetWindowExtEx) if you are using scalable modes.
3. Draw your picture using standard GDI calls.
4. Close it with CloseMetaFile.

**Step B: Wrap it in METAFILEPICT**

1. Allocate global memory (GlobalAlloc) for the structure.
2. Fill in the mm, xExt, and yExt fields.
   - ✓ *Recommendation:* Use MM_ANISOTROPIC. This gives the receiving program the most freedom to resize the image.
3. Store the Metafile Handle (hMF) inside.

**Step C: Put on Clipboard**

1. OpenClipboard -> EmptyClipboard.
2. SetClipboardData(CF_METAFILEPICT, hGlobalMemory).
3. CloseClipboard.

## III. Playing the Metafile (The Paste Side)

When your program wants to paste a drawing:

**Step A: Get the Data**

1. OpenClipboard.
2. GetClipboardData(CF_METAFILEPICT).
3. Lock the memory to read the structure.

**Step B: Setup the Environment** You cannot just play the metafile blindly. You must prepare your Device Context (DC) to match the settings of the original drawing.

1. **SaveDC:** Always save your current state so the metafile doesn't mess up your app.
2. **SetMapMode:** Read the mm field from the structure and set your DC to match it.

**Step C: Handle the Size (The Hard Part)**

- **If the mode is Fixed (e.g., MM_LOMETRIC):** The drawing has a specific physical size (e.g., 5cm x 5cm). Use the xExt and yExt values to set a **Clipping Rectangle** so the drawing doesn't bleed over your other content.
- **If the mode is Scalable (MM_ISOTROPIC / ANISOTROPIC):** The drawing is stretchy. Use SetViewportExtEx to force the drawing to fit into the specific box where the user clicked "Paste."

**Step D: Play**

1. PlayMetaFile(hdc, hMF).
2. RestoreDC.
3. Clean up handles.

## IV. Key Rules for Developers

**The Coordinate Logic**

- **MM_TEXT (Pixels):** Good for icons. Bad for printed documents.
- **MM_ISOTROPIC:** Keeps the aspect ratio (a circle stays a circle). Good for logos.
- **MM_ANISOTROPIC:** Stretches freely. Good for charts that need to fill a specific rectangle.

**Zero vs. Non-Zero Dimensions**

- If xExt is **0**, it means "I don't care about size. Stretch me to fill the whole screen."
- If xExt is **Positive**, it suggests a preferred size.

**Memory Efficiency** Always use Memory-Based Metafiles (CreateMetaFile(NULL)) for the clipboard. Do not try to pass a file path on the clipboard; it is slow and unreliable.

## V. Summary Checklist

1. **Copy:** Wrap your HMETAFILE inside a METAFILEPICT structure.
2. **Paste:** Read the Mapping Mode (mm) first.
3. **Setup:** Call SetMapMode to match the source.
4. **Scale:** Use SetViewportExtEx to map the image size to your destination rectangle.
5. **Play:** Call PlayMetaFile.

## UNDERSTANDING PREPAREMETAFILE

This function prepares the device context so a metafile can be played correctly.

It makes sure the drawing keeps the right size and shape by handling scaling and aspect ratio properly.

### I. Matching Mapping Modes:

The first thing it does is make sure the device context uses the same mapping mode as the metafile.

This is important because mapping modes control how the metafile's logical coordinates are converted into screen coordinates.

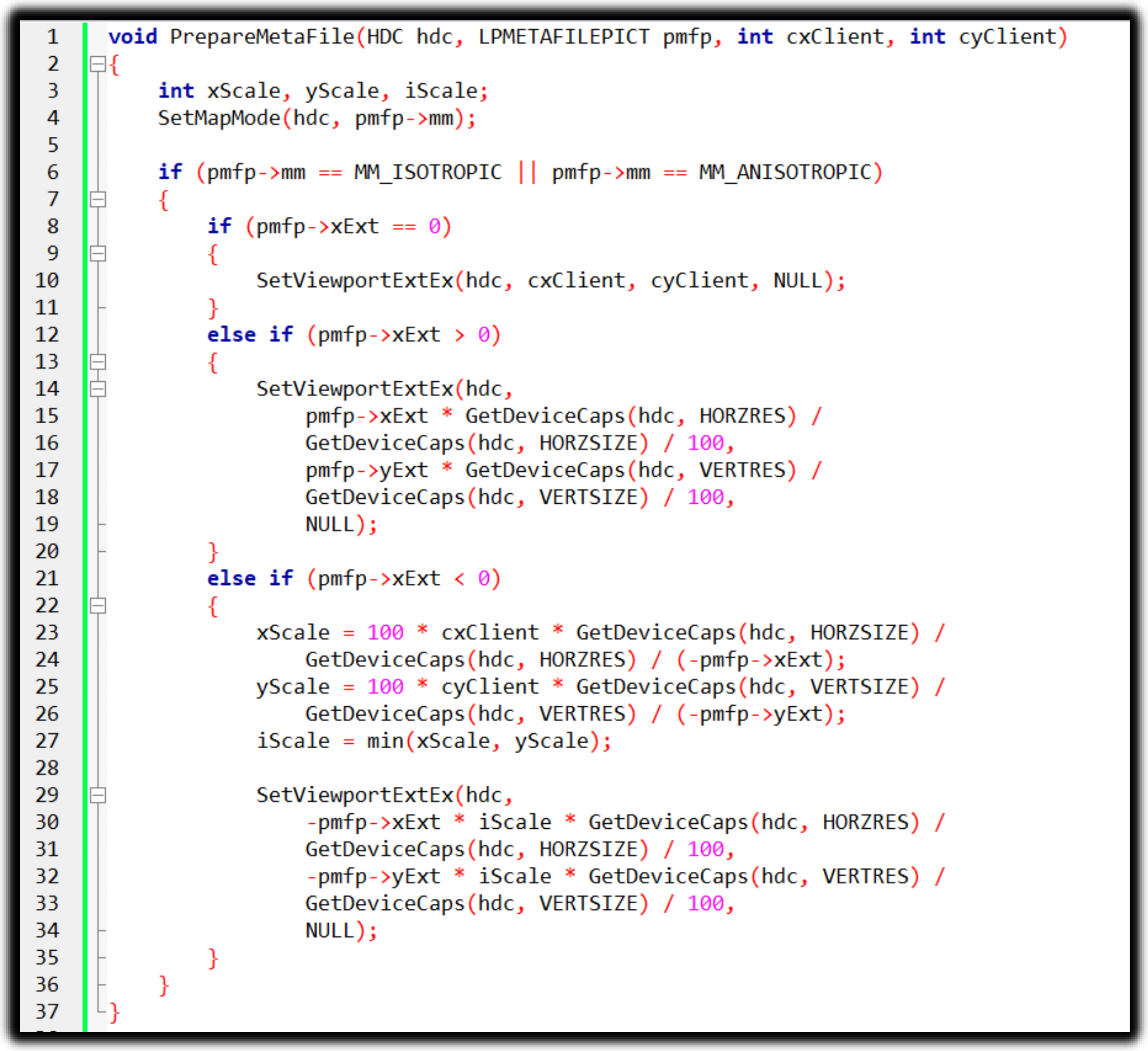
If they don't match, the drawing may appear stretched or misplaced.

### II. Handling Isotropic and Anisotropic Modes:

Special care is taken when the mapping mode is MM_ISOTROPIC or MM_ANISOTROPIC.

These modes allow more control over scaling, which helps keep the correct proportions of the drawing.

This is especially useful when the metafile needs to look consistent on different screen sizes or resolutions.

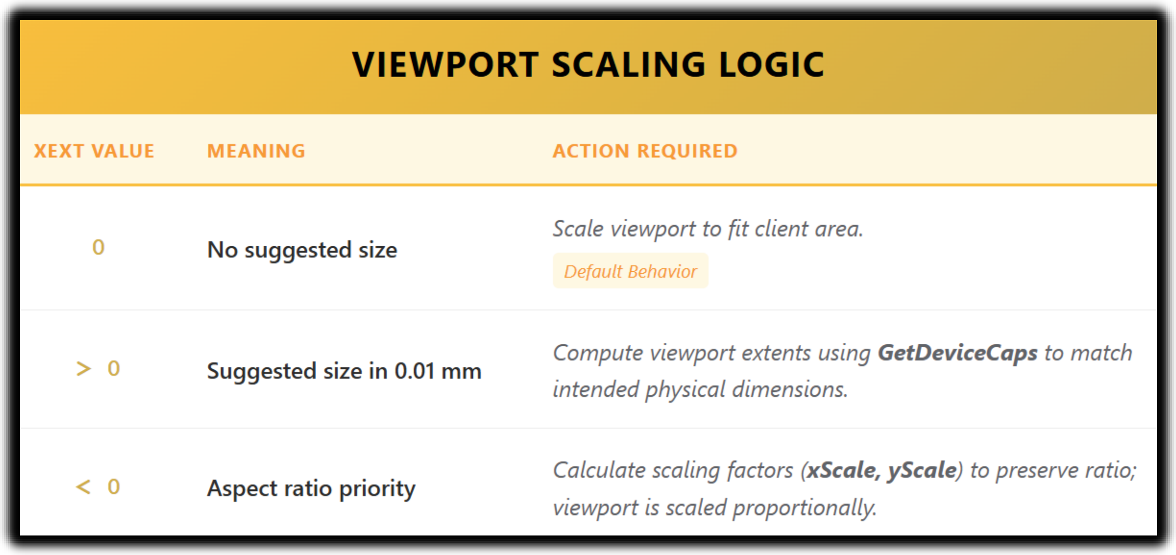


```
void PrepareMetaFile(HDC hdc, LPMETAFILEPICT pmfp, int cxClient, int cyClient)
{
    int xScale, yScale, iScale;
    SetMapMode(hdc, pmfp->mm);

    if (pmfp->mm == MM_ISOTROPIC || pmfp->mm == MM_ANISOTROPIC)
    {
        if (pmfp->xExt == 0)
        {
            SetViewportExtEx(hdc, cxClient, cyClient, NULL);
        }
        else if (pmfp->xExt > 0)
        {
            SetViewportExtEx(hdc,
                pmfp->xExt * GetDeviceCaps(hdc, HORZRES) /
                GetDeviceCaps(hdc, HORZSIZE) / 100,
                pmfp->yExt * GetDeviceCaps(hdc, VERTRES) /
                GetDeviceCaps(hdc, VERTSIZE) / 100,
                NULL);
        }
        else if (pmfp->xExt < 0)
        {
            xScale = 100 * cxClient * GetDeviceCaps(hdc, HORZSIZE) /
                GetDeviceCaps(hdc, HORZRES) / (-pmfp->xExt);
            yScale = 100 * cyClient * GetDeviceCaps(hdc, VERTSIZE) /
                GetDeviceCaps(hdc, VERTRES) / (-pmfp->yExt);
            iScale = min(xScale, yScale);

            SetViewportExtEx(hdc,
                -pmfp->xExt * iScale * GetDeviceCaps(hdc, HORZRES) /
                GetDeviceCaps(hdc, HORZSIZE) / 100,
                -pmfp->yExt * iScale * GetDeviceCaps(hdc, VERTRES) /
                GetDeviceCaps(hdc, VERTSIZE) / 100,
                NULL);
        }
    }
}
```

## Metafile Playback & Extent Handling

### I. Interpreting xExt / yExt Values

Metafiles may provide **suggested size and/or aspect ratio** using xExt and yExt from METAFILEPICT.



VIEWPORT SCALING LOGIC

| XEXT VALUE | MEANING | ACTION REQUIRED |
|---|---|---|
| 0 | No suggested size | *Scale viewport to fit client area.* *Default Behavior* |
| > 0 | Suggested size in 0.01 mm | *Compute viewport extents using **GetDeviceCaps** to match intended physical dimensions.* |
| < 0 | Aspect ratio priority | *Calculate scaling factors (**xScale, yScale**) to preserve ratio; viewport is scaled proportionally.* |

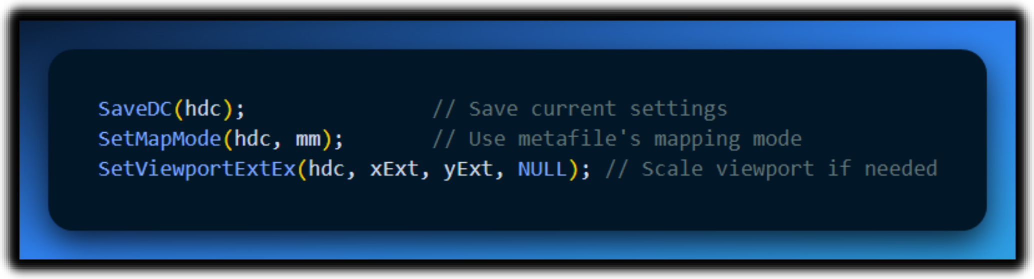
**Tip:** Always check mapping mode (mm) because it affects how logical units convert to device units.

### II. Optional Viewport Origin

- Use SetViewportOrgEx(hdc, xOrigin, yOrigin, NULL) if you need precise placement of the metafile in the client window.
- Otherwise, default origin (0,0) is fine.

### III. Playing the Metafile

Prepare device context:



```
SaveDC(hdc);                  // Save current settings
SetMapMode(hdc, mm);          // Use metafile's mapping mode
SetViewportExtEx(hdc, xExt, yExt, NULL); // Scale viewport if needed
```

Execute drawing:

```
PlayMetaFile(hdc, hMF);     // Draw metafile contents
```

Restore DC:

```
RestoreDC(hdc, -1);          // Return to previous settings
```

## IV. Releasing Resources

- **Memory unlock**: GlobalUnlock(hGlobal) – free the memory block holding the metafile data.
- **Close clipboard**: CloseClipboard() – allows other apps to access the clipboard.

**Always do this** to prevent leaks and ensure system stability.

## V. Enhanced Metafile (EMF) Considerations

- Windows **automatically converts** older metafile formats to EMF if needed.
- Clipboard exchange handles format translation for you.
- Viewport origin can still be controlled for precise placement.
- **Resource management** is the same: unlock memory, close clipboard, delete handles if created.

## VI. Key Takeaways for WinAPI Coding

- **Understand extent values** (xExt, yExt) → critical for scaling & aspect ratio.
- **Always check and set mapping mode** before playing a metafile.
- **Viewport origin control** gives precise positioning flexibility.
- **Resource cleanup is non-negotiable**: memory unlock, clipboard close, delete handles.
- EMFs make life easier — Windows handles format conversions automatically.

---

💡 **Pro Tip:** Think of xExt/yExt + mapping mode + viewport as a **recipe for translating metafile coordinates to actual pixels**. Once you nail that, rendering any metafile correctly on any display becomes straightforward.

---

# ENHANCED METAFILES (EMF) – THE 32-BIT UPGRADE 🚀

## I. Why EMFs Exist

- **32-bit Windows improvement over WMFs**. Designed to overcome **limitations of old WMFs** (like poor scaling, limited graphics support, sparse headers).

## II. Key Features

- **New Functions & Structures** → more control over drawing and playback.
- **Distinct Clipboard Format** → CF_ENHMETAFILE.
- **File Extension** → .EMF.
- **Rich Header Info**:
  - ✓ Device context settings
  - ✓ Image dimensions
  - ✓ Embedded object sizes
  - ✓ Color management
  - ✓ Thumbnail previews
  - ✓ Application-specific metadata

**Impact:** Apps can optimize playback, scale precisely, and manage advanced graphics efficiently.

## III. Compatibility

- Windows can **convert EMF ↔ WMF** for older apps.
- ⚠️ Not all EMF features survive the downgrade → paths, regions, enhanced text may be lost.

## IV. Workflow / Procedure

1. **Create** → CreateEnhMetaFile(hdc, filename, ...)
2. **Draw** → Standard GDI functions within metafile context
3. **Close** → CloseEnhMetaFile(hmf) → returns handle
4. **Play** → PlayEnhMetaFile(hdc, hmf) → renders the drawing

## V. Advantages Over WMF

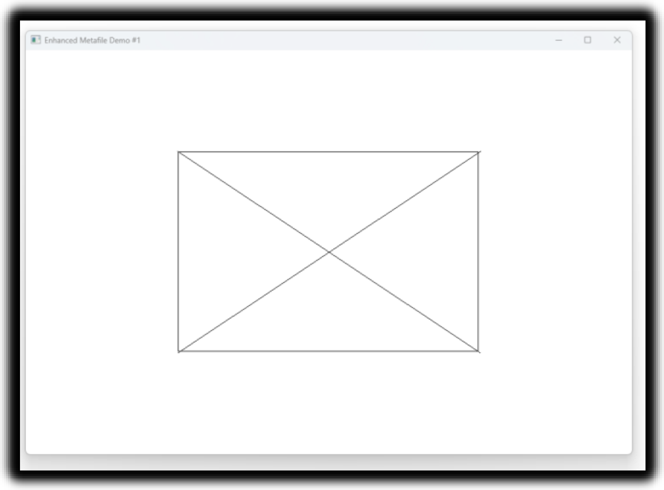
- **Advanced Graphics** → better text handling, regions, and paths
- **Scalability** → resolution-independent, no pixel distortion
- **Compactness** → smaller file sizes
- **Extensibility** → store app-specific data in the metafile

---

💡 ***Takeaway:*** EMFs = modern, flexible, precise, and smaller metafiles with richer headers. Once you switch to EMFs, you get **better scaling, better graphics, and easier clipboard integration**, while still being able to interact with older WMF-based systems if needed.

---

# EMF1.C – ENHANCED METAFILE ESSENTIALS ⚡



## I. Window Setup

- **Window Class Registration**: Sets up a window named "EMF1" with default sizes/positions.
- **Message Loop**: Keeps the program responsive to input, resizing, and system events.

## II. Creating an EMF (WM_CREATE)

- **CreateEnhMetaFile** → returns a special DC for recording drawing commands.
- **Draw Commands** → Rectangle, MoveToEx, LineTo create shapes on the metafile.
- **CloseEnhMetaFile** → finalizes the metafile and gives you a handle for playback.

## III. Playing the EMF (WM_PAINT)

- **BeginPaint / Device Context** → gets the drawing canvas.
- **Central Display** → focus playback in a rectangle within the window.
- **PlayEnhMetaFile** → replays drawing commands.
- **Stretching** → GDI automatically scales the image to fit the rectangle (can distort).

## IV. Cleaning Up (WM_DESTROY)

- **DeleteEnhMetaFile** → releases memory used by the metafile.
- Ensures **no memory leaks** and smooth exit.

## V. Core Concepts

- **Coordinate Independence**: Only the **relative positions** of shapes matter; absolute numbers can be adjusted.
- **Image Stretching**: Flexibility vs. fidelity — stretching can distort shapes but allows fitting into any rectangle.
- **Preserving Aspect Ratio / Size**: Important for precise visuals (e.g., technical drawings, police sketches).
- **Line-Rectangle Mismatch**: Windows may misalign lines vs. rectangle corners; a known quirk to fix if needed.

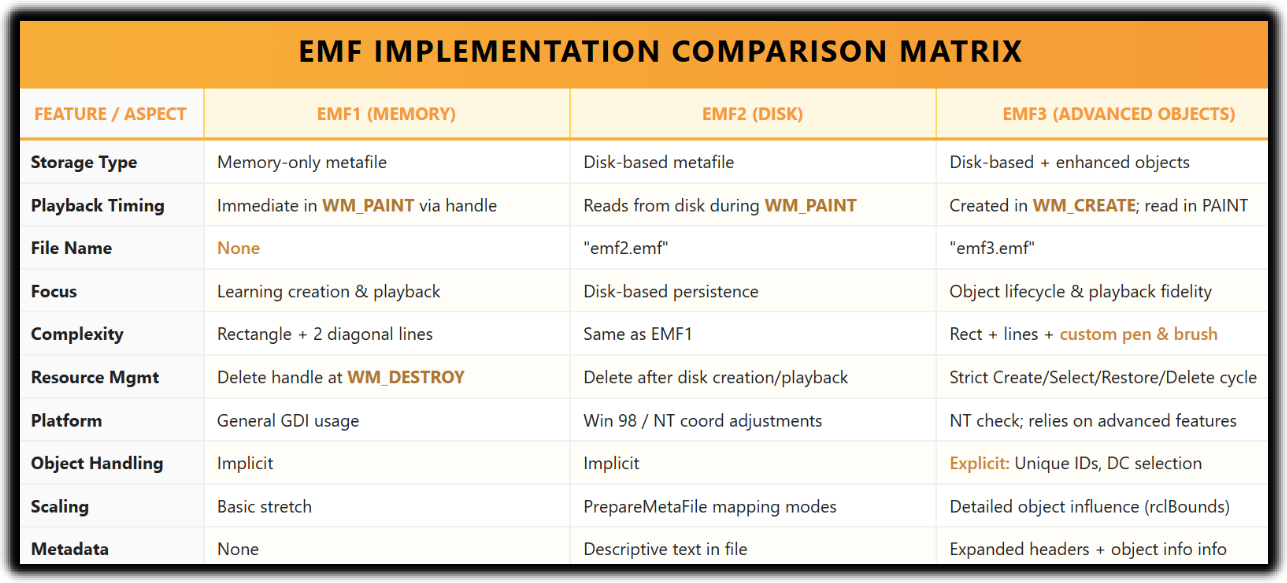
## VI. Advanced Takeaways

- **EMFs vs. WMFs**: EMFs are more robust, support advanced graphics, and scale better.
- **Flexible Playback**: You can place EMFs in different rectangles without losing relative geometry.
- **GDI Role**: Handles both drawing and scaling, making metafile usage practical and versatile.
- **Cross-Platform Note**: Other systems use formats like SVG; EMFs are mainly Windows-specific.

## 💡 TLDR

- EMFs **record drawing commands** into a device-independent format.
- CreateEnhMetaFile → draw → CloseEnhMetaFile → store handle → PlayEnhMetaFile to display.
- Coordinates are relative; you can scale/stretch to fit.
- Balance **accuracy** (aspect ratio, exact size) vs **flexibility** (fit any rectangle).
- Always **clean up resources** with DeleteEnhMetaFile.

## EMF1 VS EMF2 VS EMF3 – CORE DIFFERENCES & LESSONS 🔥



EMF IMPLEMENTATION COMPARISON MATRIX

| FEATURE / ASPECT | EMF1 (MEMORY) | EMF2 (DISK) | EMF3 (ADVANCED OBJECTS) |
|---|---|---|---|
| **Storage Type** | Memory-only metafile | Disk-based metafile | Disk-based + enhanced objects |
| **Playback Timing** | Immediate in **WM_PAINT** via handle | Reads from disk during **WM_PAINT** | Created in **WM_CREATE**; read in PAINT |
| **File Name** | None | "emf2.emf" | "emf3.emf" |
| **Focus** | Learning creation & playback | Disk-based persistence | Object lifecycle & playback fidelity |
| **Complexity** | Rectangle + 2 diagonal lines | Same as EMF1 | Rect + lines + **custom pen & brush** |
| **Resource Mgmt** | Delete handle at **WM_DESTROY** | Delete after disk creation/playback | Strict Create/Select/Restore/Delete cycle |
| **Platform** | General GDI usage | Win 98 / NT coord adjustments | NT check; relies on advanced features |
| **Object Handling** | Implicit | Implicit | Explicit: Unique IDs, DC selection |
| **Scaling** | Basic stretch | PrepareMetaFile mapping modes | Detailed object influence (rclBounds) |
| **Metadata** | None | Descriptive text in file | Expanded headers + object info info |

**NB:** *Find the html for this table in this folder.*

## I. Key Best Practices for Metafile Resource Management✅

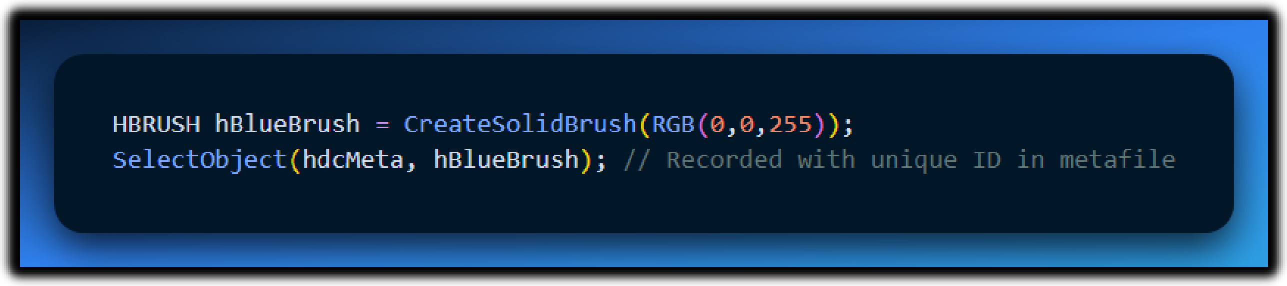
1. **Delete Objects After Use**
   - ✓ Always call DeleteObject on pens, brushes, and other GDI objects after they're no longer needed.
   - ✓ Prevents leaks and reduces metafile size during playback.
2. **Restore Default Objects**
   - ✓ After drawing with custom pens/brushes, restore stock objects via SelectObject.
   - ✓ Ensures the DC is ready for future operations and prevents conflicts.
3. **Limit Object Creation**
   - ✓ Avoid creating unnecessary pens or brushes. Reuse them whenever possible.
   - ✓ Reduces memory usage and file size.
4. **Handle Disk vs Memory**
   - ✓ Use disk-based metafiles when you need to inspect, share, or reuse the file later.
   - ✓ Memory-based metafiles are fast for transient rendering but lost on program exit.
5. **Prepare for Different Mapping Modes**
   - ✓ For MM_ISOTROPIC / MM_ANISOTROPIC, calculate scaling factors to preserve aspect ratio.
   - ✓ For other modes, use clipping rectangles based on xExt / yExt.

---

## II. Using Unique Identifiers for Non-Default Objects

- **What They Do:** Each custom pen/brush/etc. gets a unique ID in the metafile.
- **How to Use:** During playback, SelectObject(hdc, objectID) lets you choose that exact object. Change properties, reuse objects, or selectively draw parts of the metafile.

**Example:**



```
HBRUSH hBlueBrush = CreateSolidBrush(RGB(0,0,255));
SelectObject(hdcMeta, hBlueBrush); // Recorded with unique ID in metafile
```

Later, you can reuse that exact brush during playback or for another metafile section without recreating it.

---

## III. Stock Objects in Metafiles

- Predefined objects, identified by **high-bit-set IDs** in EMR_SELECTOBJECT records.
- Examples:
  - ✓ **Stock Pens:** Black pen, default width
  - ✓ **Stock Brushes:** White brush, hollow brush
  - ✓ **Stock Fonts:** ANSI_FIXED_FONT, SYSTEM_FONT, DEFAULT_GUI_FONT
- **Benefit:** Saves memory, reduces metafile size, no need to explicitly create them.

---

## IV. EMF3 Structural Insights

1. EMF3 stores **pen & brush definitions** explicitly (EMR_EXTCREATEPEN, EMR_CREATEBRUSHINDIRECT).
2. Selection and deletion are recorded (EMR_SELECTOBJECT, EMR_DELETEOBJECT) → ensures **full playback fidelity**.
3. Hex dumps show the **exact sequence of GDI calls**, giving insight into:
   - ✓ Object creation order.
   - ✓ Drawing commands.
   - ✓ Resource cleanup.
4. rclBounds adjusts for pen thickness → affects drawing boundary.
5. nHandles shows how many objects are stored → increased in EMF3 vs EMF2 due to extra objects.

---

## V. Summary Lessons

- **EMF1:** Simple, in-memory creation and immediate playback; good for learning basics.
- **EMF2:** Introduces disk persistence; good for inspecting file contents and learning storage concepts.
- **EMF3:** Shows **real-world metafile management** — custom objects, proper selection/deletion, mapping modes, scaling, and object IDs.

💡 **Big Picture:** As you move from EMF1 → EMF3, the focus shifts from **basic drawing** to **structured resource handling and object lifecycle**, which is essential for production-level GDI graphics programming.

---

# EMF4 PROGRAM – REVAMPED OVERVIEW ✨

## I. Purpose & Key Features

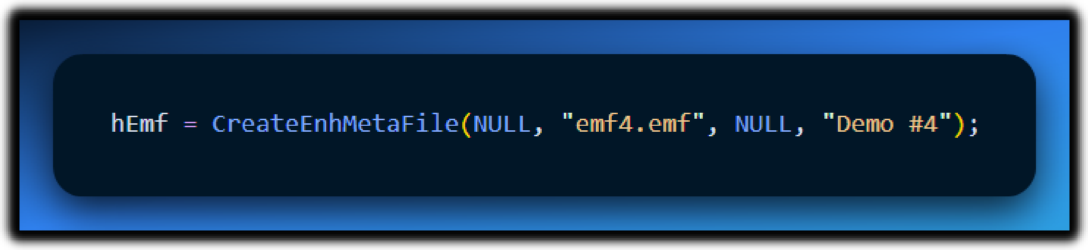
- **Bitmap Integration:** EMF4 demonstrates how bitmaps can be embedded into enhanced metafiles, expanding the range of graphics beyond simple shapes.
- **Disk-Based Persistence:** The metafile is created and saved to disk (emf4.emf), allowing inspection and delayed playback.
- **Playback Flexibility:** Supports scaling, repositioning, and consistent rendering on different devices using device-independent bitmaps (DIBs).

## II. Window Creation & Initialization

- Registers a standard window class and creates the main window.
- Sets up the environment for drawing and playback.

## III. WM_CREATE – Capturing the Bitmap

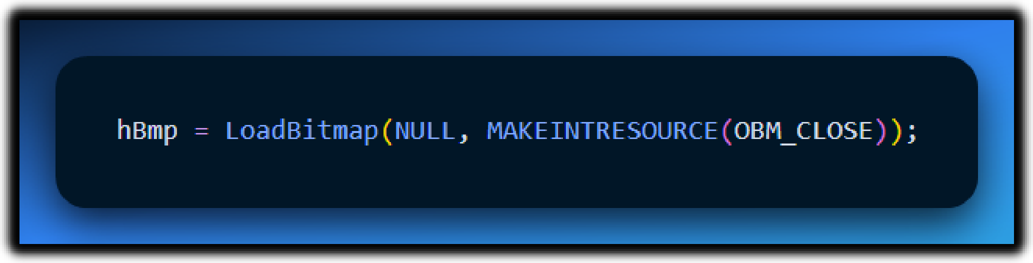
**Metafile Creation:**



```
hEmf = CreateEnhMetaFile(NULL, "emf4.emf", NULL, "Demo #4");
```

Prepares an enhanced metafile to record drawing commands.

**Load Bitmap from System Resources:**



```
hBmp = LoadBitmap(NULL, MAKEINTRESOURCE(OBM_CLOSE));
```

Retrieves the standard “close button” bitmap.

**Bitmap Information & Memory DC:**

- GetObject obtains width, height, and other properties.
- CreateCompatibleDC creates a temporary memory DC for manipulating the bitmap before recording.

**Select & Transfer Bitmap:**

- Select bitmap into memory DC with SelectObject.
- Copy it into the metafile DC at (100, 100) with size 100x100 using StretchBlt.
- This action is **recorded into the metafile**, including scaling and placement.

**Resource Cleanup:**

- Delete memory DC: DeleteDC.
- Delete bitmap object: DeleteObject.

**Finalize Metafile:**

- Close with CloseEnhMetaFile.
- Delete handle from memory using DeleteEnhMetaFile. The file on disk still contains all content.

## IV. WM_PAINT – Recreating the Scene

1. Prepare canvas with BeginPaint.
2. Retrieve client area dimensions using GetClientRect.
3. Center the metafile in the window for balanced display.
4. Load metafile from disk: GetEnhMetaFile("emf4.emf").
5. Render metafile to DC with PlayEnhMetaFile, including the embedded bitmap.
6. Release resources: DeleteEnhMetaFile.
7. End painting: EndPaint.

## V. Core Takeaways

- **Metafiles are versatile:** Can capture both GDI commands and embedded bitmaps.
- **Memory DCs matter:** Act as intermediate canvases for bitmap manipulation before recording into a metafile.
- **Persistence:** Content remains in disk metafile even after deletion from memory.
- **Playback flexibility:** Size, position, and device-independent scaling can be adjusted during WM_PAINT.

## VI. Handling Bitmaps – GDI Strategies

**Indirect Objects:** Functions like LoadBitmap and CreateCompatibleDC aren't recorded directly. Instead:

- Bitmap data is stored as an **indirect object** inside the metafile.
- Dimensions, color depth, and other attributes are saved for accurate playback.

**Playback Methods:**

- **Direct:** StretchDIBits uses the embedded DIB data directly.
- **Reconstruction:** CreateDIBitmap builds a device-dependent bitmap (DDB) and then uses StretchBlt for rendering.

## VII. Metafile Structure

- **Header:** Signature, bounds, EMF version, descriptive text ("Demo #4").
- **StretchBlt Record:** References bitmap indirect object, source & destination rectangles, ROP code.
- **Bitmap Data:** Stored as a DIB (device-independent bitmap), self-contained for portability.
- **EOF Record:** Marks end of metafile.

## VIII. Key Lessons for Graphics Programming

- **Efficiency:** Record only essential drawing commands for playback; avoid storing the entire bitmap handling process.
- **Portability:** DIBs allow bitmaps to play back correctly across different devices and contexts.
- **Resource Management:** Clean up temporary DCs and objects to prevent leaks.
- **Playback Strategies:** Understand GDI's dual approach to rendering bitmaps for both speed and device compatibility.

---

## ✅ Summary:

EMF4 elevates metafiles to handle **real-world graphics** like bitmaps efficiently, emphasizing **playback-centric design, portability, and proper resource management**. Think of it as EMF3 plus **bitmap embedding and sophisticated GDI handling**.

## CLOSING THE EMF SERIES: PROGRAMS 5, 6, AND 7 🎬

This final group of programs explores what you can really *do* with Enhanced Metafiles beyond basic playback. The focus moves from simple rendering to control, transformation, and embedding.

---

### I. EMF5 — Controlled Playback (Enumeration)

EMF5 renders the same output as EMF3, but it uses a different approach.

Instead of playing the metafile in one call, it enumerates the metafile record by record.

Each drawing instruction is processed individually through a callback function. During enumeration, each record is executed using PlayEnhMetaFileRecord.

This method gives fine-grained control over playback. You can inspect records, skip them, or decide when and where they are rendered.

The visual result is the same as EMF3. The difference is control.

**Key idea:**
EMF5 proves that enumeration is about *how* you play a metafile, not *what* it draws.

---

### II. EMF6 — Transforming Records During Playback

EMF6 takes enumeration a step further by modifying drawing commands while the metafile is being played.

As each record is enumerated, the program checks whether it is a rectangle drawing command. If it is, the record is copied into temporary memory. The copy's record type is changed from rectangle to ellipse. The modified copy is then played instead of the original.

The original metafile is never changed. Only the copied records are altered.

This approach preserves the integrity of the metafile while allowing creative transformations during rendering.

EMF6 also introduces the importance of the GDI handle table. GDI objects such as pens and brushes are tracked internally and reused correctly during playback.

**Key idea:**
EMF6 shows how enumeration can be used to *transform graphics safely*.

## III. EMF7 — Embedding Graphics into a New Metafile

EMF7 is different from everything before it.

Instead of modifying playback, it creates a brand-new metafile that embeds new graphics into existing content.

The program creates a new metafile device context and enumerates the original metafile into it. While enumeration is happening, extra drawing commands are injected directly into the new metafile. In this case, a green ellipse is added when a rectangle record is encountered.

The original metafile remains untouched. A new metafile is saved to disk containing both the original drawings and the embedded graphics.

During enumeration, non-visual records such as the header and EOF are skipped. GDI state is preserved by saving and restoring pens and brushes.

When the window repaints, the program simply plays the newly created metafile.

**Key idea:**
EMF7 demonstrates **non-destructive metafile composition**.

---

## IV. Big Picture Summary 🧠

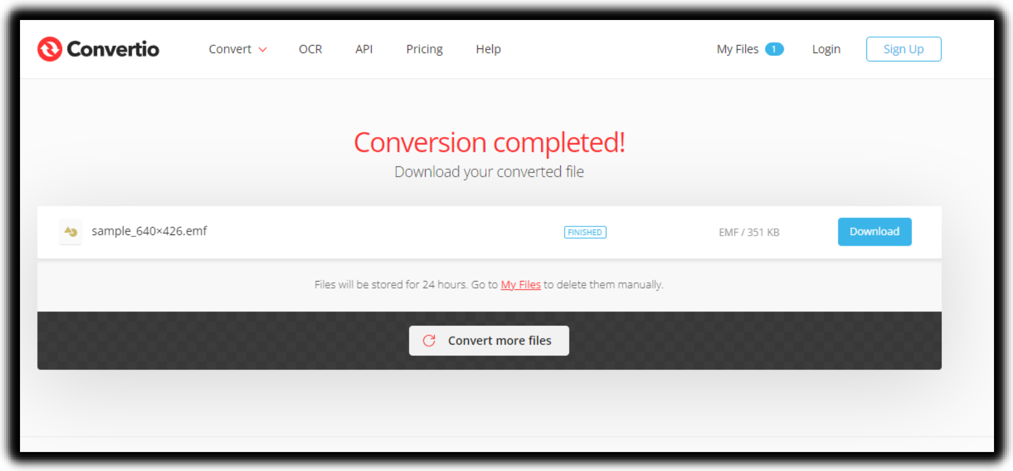
EMF3 plays a metafile all at once.
EMF5 plays it record by record.
EMF6 modifies records during playback.
EMF7 writes new content into a new metafile.

Enumeration is the foundation behind all advanced EMF techniques. Once you understand it, you can inspect, filter, transform, and even generate new metafiles programmatically.

That’s the real power of EMF.

## EMFVIEW PROGRAM

Note the first thing I did was convert the regular .bmp file to .emf file using:



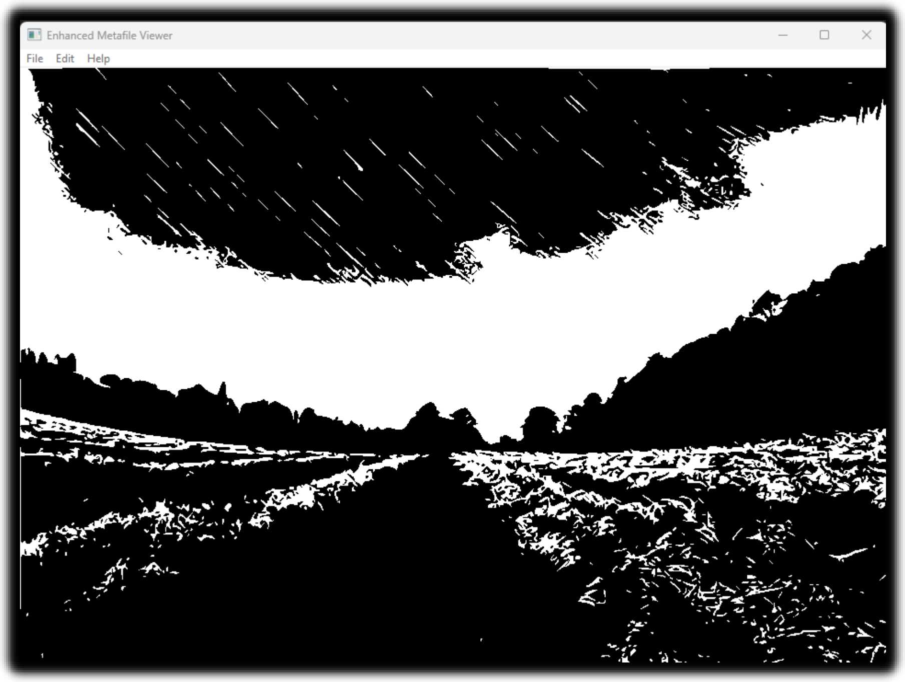
Conversion completed!
Download your converted file

sample_640×426.emf FINISHED EMF / 351 KB Download

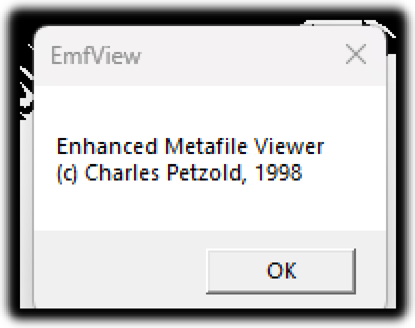
Files will be stored for 24 hours. Go to My Files to delete them manually.

...or any other tool online. Loaded as a regular .bmp

Loaded as a .emf file:



The about:



EmfView

Enhanced Metafile Viewer
(c) Charles Petzold, 1998

## What EMFVIEW *Really* Is (No Fluff)

Strip all the ceremony away and EMFVIEW is just this:

**EMFVIEW = A Windows app that loads an EMF, shows it, lets you copy/save/print it, and displays its metadata.**

That's it. Nothing mystical. No hidden wizardry.

Everything else is just Windows plumbing.

## Core Responsibilities (Plain English)

### I. Open & Display an EMF

- Uses GetEnhMetaFile
- Stores the handle
- Calls PlayEnhMetaFile in WM_PAINT
- Stretches it to fit the window

👉 **Viewer logic**

### II. Save / Duplicate an EMF

- Uses CopyEnhMetaFile
- Writes a new .emf file

👉 **File I/O wrapper**

### III. Print an EMF

- Opens printer dialog (PrintDlg)
- Gets printer DC
- Starts a document (StartDoc → StartPage)
- Plays the metafile into printer bounds
- Ends page and doc

👉 **Same drawing, different DC**

This is the big WinAPI lesson:

**Screen DC and Printer DC are just output targets.**

## IV. Clipboard Support

- Copy / cut uses CopyEnhMetaFile
- Paste pulls EMF from clipboard
- Delete clears the handle

👉 **Standard Windows clipboard behavior**

## V. Properties Dialog

- Calls GetEnhMetaFileHeader
- Dumps:
    - ✓ Bounds
    - ✓ Frame
    - ✓ Resolution
    - ✓ Record count
    - ✓ Handle count
    - ✓ Palette entries

👉 **Metadata inspection**

This is literally just *reading the header*.

## VI. Palette Handling (Old-school but important)

- Extracts palette from EMF
- Selects & realizes it
- Responds to:
    - ✓ WM_QUERYNEWPALETTE
    - ✓ WM_PALETTECHANGED

👉 **Color correctness on limited hardware.**

Not cool, but historically critical.

## VII. The Printing “Problem” You Noticed Is REAL

You caught something important 👀

Thin vector lines:

- Look fine on screen
- Disappear on high-DPI printers

That’s not a bug. That’s physics.

**Fix:** use thicker pens when printing.
EMF doesn’t magically know your intent.

That’s pro-level awareness, by the way.

## VIII. The Scaling Truth (This Is the Key Insight)

Metafiles scale well **because they are instructions**, not pixels.

But:

- Embedded bitmaps don’t scale infinitely
- Raster fonts don’t scale cleanly
- Aspect ratio *can* be destroyed if you’re careless

That’s why:

- Header info matters
- Bounding rectangles matter

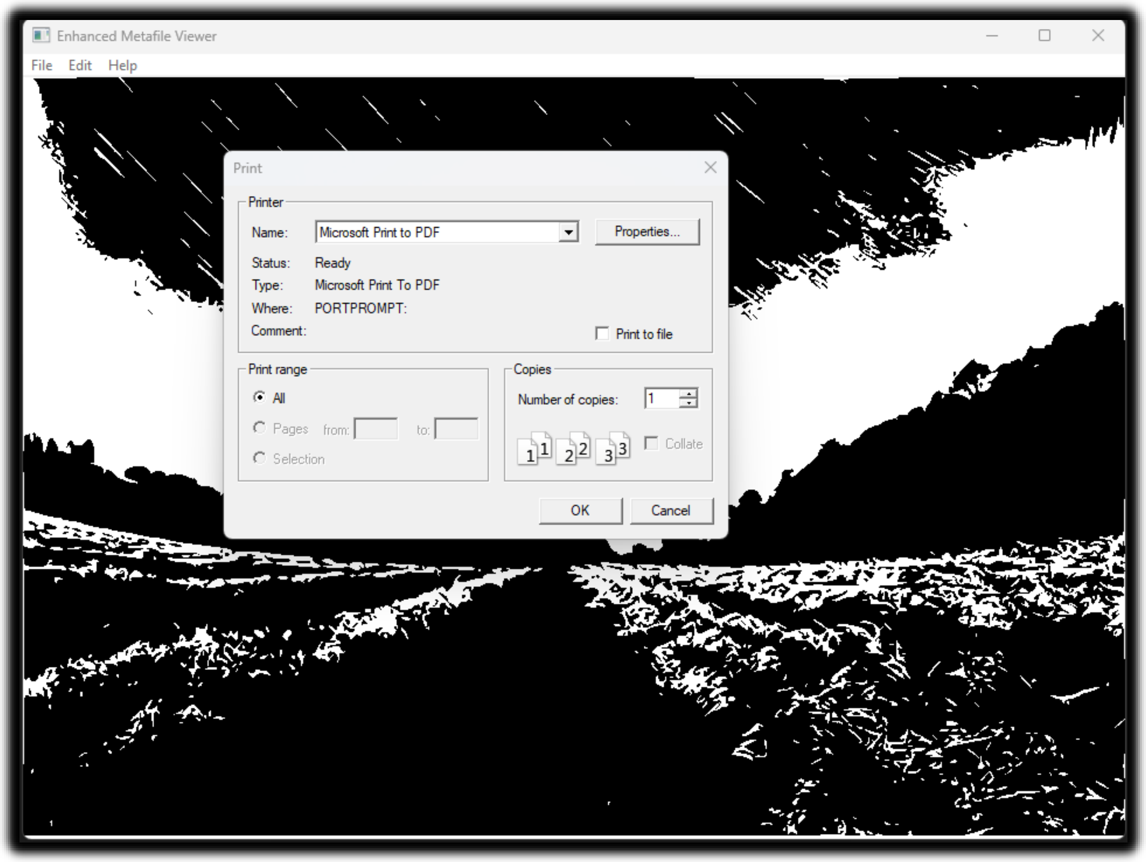
PlayEnhMetaFile(hdc, hemf, &rect);
That rectangle **is the contract**.

## IX. Ultra-Compressed

If I had to reduce EMFVIEW to **10 lines of notes**, it'd be this:

- EMFVIEW loads, displays, prints, and copies EMF files.
- Uses GetEnhMetaFile and PlayEnhMetaFile.
- Printing uses the same playback logic on a printer DC.
- Clipboard uses CopyEnhMetaFile.
- Properties come from GetEnhMetaFileHeader.
- Scaling is controlled by the bounding rectangle.
- Vector graphics scale cleanly; bitmaps don't.
- Thin lines may vanish on high-DPI printers.
- Palettes ensure color accuracy on limited systems.
- Same drawing logic, different output targets.

That's it. No overthinking.



Printer

Name: Microsoft Print to PDF

Status: Ready

Type: Microsoft Print To PDF

Where: PORTPROMPT:

Comment:

[ ] Print to file

Print range

[✓ All]

Pages from: to:

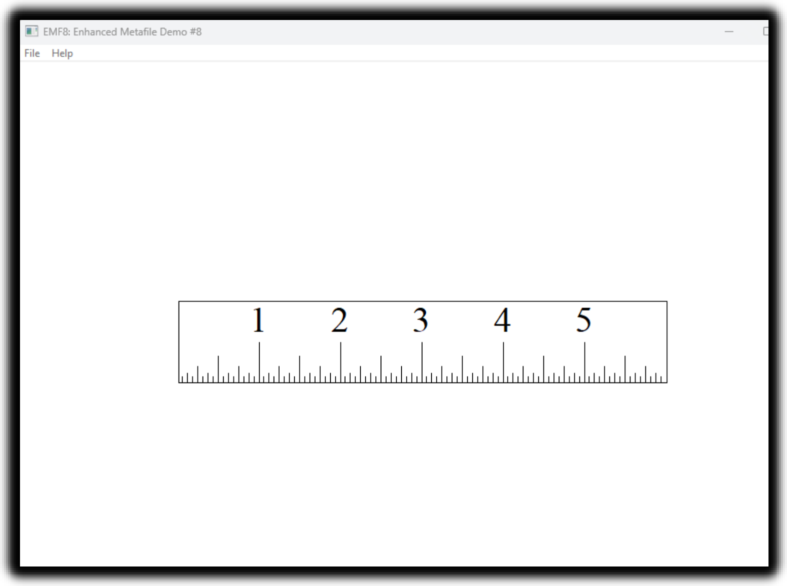
Selection

Copies

Number of copies: 1

[ ] Collate

## EMF8 / EMF.C — THE SIGNAL



## What the program actually demonstrates

Accurate, device-independent rendering of a 6-inch ruler using EMFs.
Nothing more. Nothing less.

**Core Idea:** EMFs store drawing commands, not pixels, so correct scaling depends on which header rectangle you use and which device was the reference.

That's it. That's the chapter.

## EMF8.C — What Matters

### I. What EMF8 does

- Creates an **EMF containing a ruler**
- Uses the **screen DC as the reference device**
- Plays the EMF on:
    - ✓ the window
    - ✓ the printer

## II. Why this is interesting

- The ruler **looks fine on screen**
- The ruler **prints at the wrong physical size**

👉 That's not a bug. That's the lesson.

## III. The real problem

- EMF8 uses **rclBounds**
- rclBounds = **pixel-based**
- Pixels depend on the reference device (screen)

So, when printed on a 300 DPI printer:

**6 inches becomes ~1⅓ inches**

Correct behavior. Wrong assumption.

## IV. EMF Header — Only Two Fields Matter Here

❌ **rclBounds**

- Units: **pixels**
- Device-dependent
- Fine for screen display
- **Bad for physical accuracy**

✅ **rclFrame**

- Units: **0.01 millimeters**
- Device-independent
- **Correct for printing and real-world measurements**

This distinction is the entire chapter.

## EMF.C — Why It Exists

EMF.C is just a **shell app**:

- Window creation
- Message loop
- WM_PAINT → calls PaintRoutine
- IDM_PRINT → prints using the same drawing logic

It's infrastructure, not the lesson.

## The Real WinAPI Lesson (This Is the Gold)

### I. Device Contexts Lie (Politely)

A DC always reflects **its device**.
If you assume pixels = inches, you will get burned.

### II. Metafiles Are Only as Accurate as Their Reference DC

When you create an EMF:

- The **reference DC defines the math**
- Screen DC ≠ Printer DC

### III. Use the Right Rectangle for the Job

- UI preview → rclBounds
- Physical accuracy → rclFrame

If it must measure correctly in the real world:

***Never trust pixels.***

*EMF8 demonstrates why using rclBounds (pixel-based) causes incorrect physical sizing when printing. EMF headers contain rclFrame (0.01 mm units), which must be used for device-independent, real-world accurate rendering (fixed in EMF9). The reference DC used during EMF creation determines how header dimensions are interpreted.*

Done. That's senior-level notes.

## EMF9 PROGRAM

```
// Program Title: Enhanced Metafile Demo #9
// Author: Charles Petzold, 1998

// Include necessary headers
#include <windows.h>
#include <string.h>

// Define window class and title
TCHAR szClass[] = TEXT("EMF9");
TCHAR szTitle[] = TEXT("EMF9: Enhanced Metafile Demo #9");

// Function to create the window
void CreateRoutine(HWND hwnd) {
    // Implementation not provided in the code snippet
    // This function would typically handle window creation, initialization, etc.
}

// Function to paint the window
void PaintRoutine(HWND hwnd, HDC hdc, int cxArea, int cyArea) {
    ENHMETAHEADER emh;
    HENHMETAFILE hemf;
    int cxMms, cyMms, cxPix, cyPix, cxImage, cyImage;
    RECT rect;

    // Get device characteristics
    cxMms = GetDeviceCaps(hdc, HORZSIZE);
    cyMms = GetDeviceCaps(hdc, VERTSIZE);
    cxPix = GetDeviceCaps(hdc, HORZRES);
    cyPix = GetDeviceCaps(hdc, VERTRES);

    // Load an enhanced metafile from a file
    hemf = GetEnhMetaFile(TEXT("..\\emf8\\emf8.emf"));

    // Retrieve the header information of the metafile
    GetEnhMetaFileHeader(hemf, sizeof(emh), &emh);

    // Calculate the image size in pixels based on device characteristics
    cxImage = emh.rclFrame.right - emh.rclFrame.left;
    cyImage = emh.rclFrame.bottom - emh.rclFrame.top;
    cxImage = cxImage * cxPix / cxMms / 100;
    cyImage = cyImage * cyPix / cyMms / 100;

    // Calculate the position to center the image in the window
    rect.left = (cxArea - cxImage) / 2;
    rect.right = (cxArea + cxImage) / 2;
    rect.top = (cyArea - cyImage) / 2;
    rect.bottom = (cyArea + cyImage) / 2;

    // Play the enhanced metafile on the specified rectangle
    PlayEnhMetaFile(hdc, hemf, &rect);

    // Delete the metafile handle to free resources
    DeleteEnhMetaFile(hemf);
}
```

## EMF9 → EMF13

### I. The Big Picture (Applies to All)

All these programs explore **one problem**:

**How to display and print a ruler metafile accurately across devices, resolutions, and layouts.**

Everything else is just **a variation knob**:

- which header rectangle you trust
- whether you preserve aspect ratio
- whether you use mapping modes
- whether scaling is manual or logical

### II. EMF9 — Correct Physical Size

**What it adds**

- Fixes EMF8's printing error
- Uses **rclFrame (0.01 mm units)** instead of rclBounds

**Core lesson**

- rclFrame = device-independent
- Correct choice for **real-world measurements**

**One-liner**

EMF9 renders the ruler at the correct physical size on both screen and printer by scaling using rclFrame.

## III. EMF10 — Preserve Aspect Ratio

**What it adds**

- Keeps the ruler’s **6:1 aspect ratio**
- Fits the ruler inside the available client area
- Centers it visually

**Core lesson**

- Scaling ≠ stretching
- Aspect ratio must be enforced manually

**One-liner**

EMF10 scales the ruler to fit the client area while preserving aspect ratio and centering the image.

## IV. EMF11 — Mapping Modes Experiment (and Warning)

**What it adds**

- Tries to draw the ruler in **inches** using MM_LOENGLISH
- Embeds mapping mode into EMF creation

**Core lesson**

- Mapping modes + metafiles = fragile
- Results vary across devices

**One-liner**

EMF11 demonstrates that using mapping modes during metafile creation can lead to inconsistent results across devices.

## EMF12 — Mapping Modes, Carefully Applied

**What it adds**

- Uses mapping modes more deliberately:
  - ✓ Creation: logical units
  - ✓ Playback: MM_HIMETRIC (0.01 mm)

**Core lesson**

- Mapping modes can work
- But they require strict discipline and testing

**One-liner**

EMF12 shows a controlled use of mapping modes to achieve device-independent playback, while highlighting their complexity.

## EMF13 — Playback Mapping Only

**What it adds**

- Does **not** create the metafile.
- Uses mapping mode **only during playback.**
- Reads size from EMF header.

**Core lesson**

- Creation mapping ≠ playback mapping
- Playback mapping only affects **positioning and scaling**, not image size stored in the EMF

**One-liner**

EMF13 demonstrates that mapping modes during playback control placement and scaling, not the metafile's intrinsic size.

## The Rules (This Is What You Actually need)

### Rule 1 — Creation vs Playback

- **Creation mapping mode** → affects EMF header size
- **Playback mapping mode** → affects destination rectangle only

### Rule 2 — Physical Accuracy

- Printing or measurement → use rclFrame
- UI preview only → rclBounds is fine

### Rule 3 — Aspect Ratio Is Your Job

- GDI won't protect it
- You must calculate scaling manually

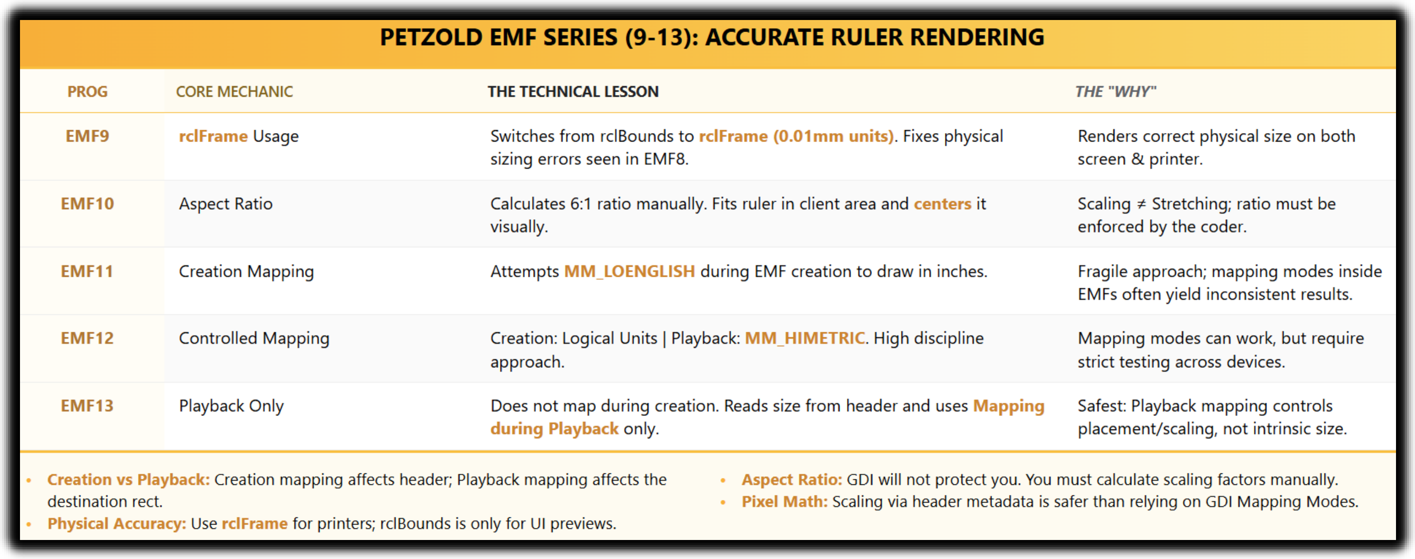
### Rule 4 — Mapping Modes Are Optional, Not Required

- Pixel math + header data = safest
- Mapping modes add elegance **and risk**

EMF9 to EMF13 progressively refine accurate ruler rendering across devices.

They demonstrate correct use of rclFrame for physical accuracy, aspect-ratio-preserving scaling, and the risks and limitations of mapping modes during metafile creation and playback.

Pixel-based scaling using header metadata remains the most reliable approach. That's it. It replaces **everything** we over-explained in the old notes.



**PETZOLD EMF SERIES (9-13): ACCURATE RULER RENDERING**

| PROG | CORE MECHANIC | THE TECHNICAL LESSON | THE "WHY" |
|---|---|---|---|
| EMF9 | rclFrame Usage | Switches from rclBounds to rclFrame (0.01mm units). Fixes physical sizing errors seen in EMF8. | Renders correct physical size on both screen & printer. |
| EMF10 | Aspect Ratio | Calculates 6:1 ratio manually. Fits ruler in client area and centers it visually. | Scaling ≠ Stretching; ratio must be enforced by the coder. |
| EMF11 | Creation Mapping | Attempts MM_LOENGLISH during EMF creation to draw in inches. | Fragile approach; mapping modes inside EMFs often yield inconsistent results. |
| EMF12 | Controlled Mapping | Creation: Logical Units \| Playback: MM_HIMETRIC. High discipline approach. | Mapping modes can work, but require strict testing across devices. |
| EMF13 | Playback Only | Does not map during creation. Reads size from header and uses Mapping during Playback only. | Safest: Playback mapping controls placement/scaling, not intrinsic size. |

- **Creation vs Playback:** Creation mapping affects header; Playback mapping affects the destination rect.
- **Physical Accuracy:** Use rclFrame for printers; rclBounds is only for UI previews.
- **Aspect Ratio:** GDI will not protect you. You must calculate scaling factors manually.
- **Pixel Math:** Scaling via header metadata is safer than relying on GDI Mapping Modes.