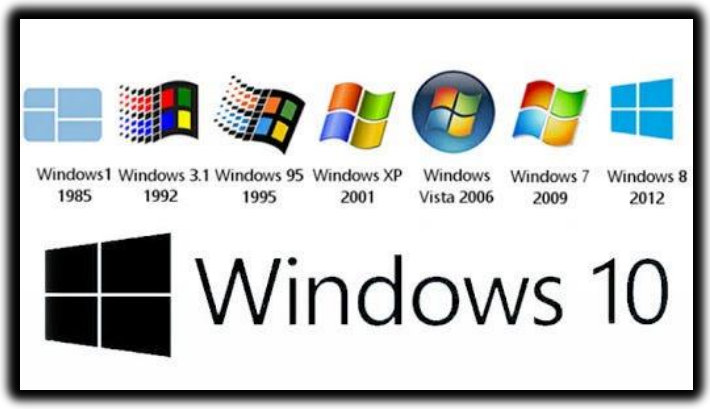
CHAPTER 22 – SOUND & MUSIC IN WINDOWS

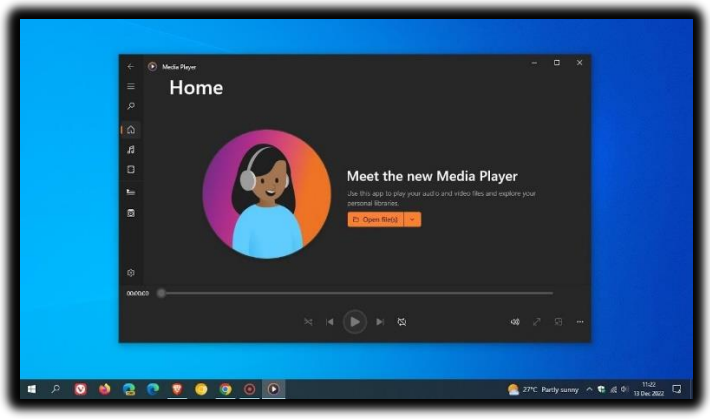
History & Context

- **Early Multimedia:** Introduced as **Multimedia Extensions** in Windows 3.0 (1991), fully integrated in **Windows 3.1 (1992)**.
- **Hardware Adoption:** Sound cards, CD-ROM drives, and video support were rare in early 90s, now standard.
- **Impact:** Multimedia changed Windows from a **text & number platform** to a **rich media platform**—audio, video, games, and creativity became part of the OS.

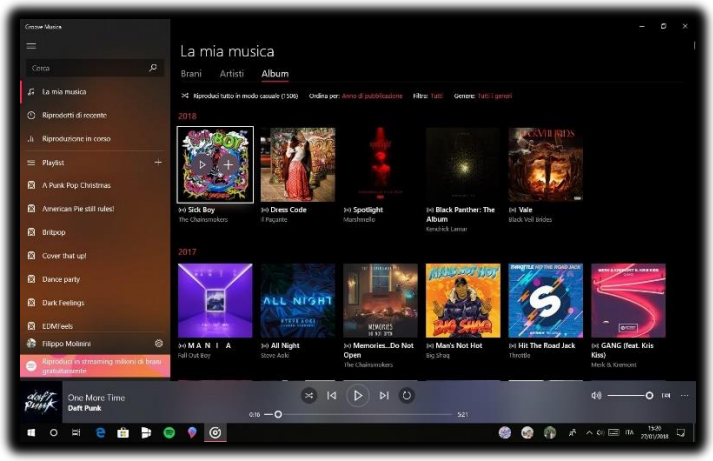
2. Built-in Windows 10 Tools



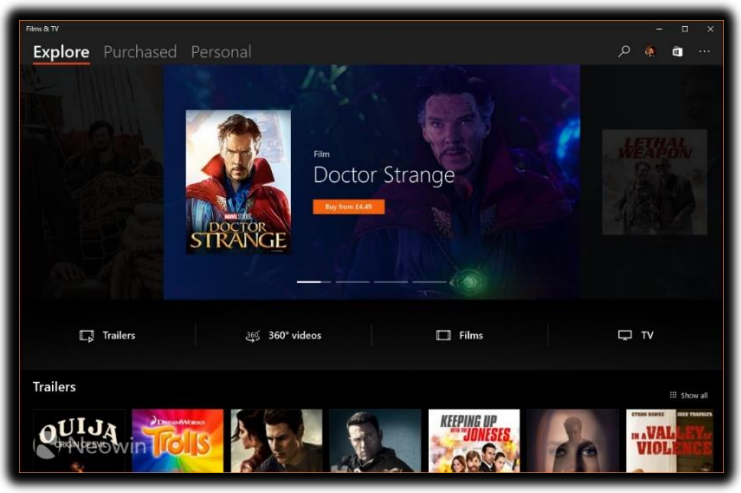
Windows Media Player: Play audio/video files, manage playlists, organize libraries.



Groove Music: Stream music, play local files, create playlists, discover new music.

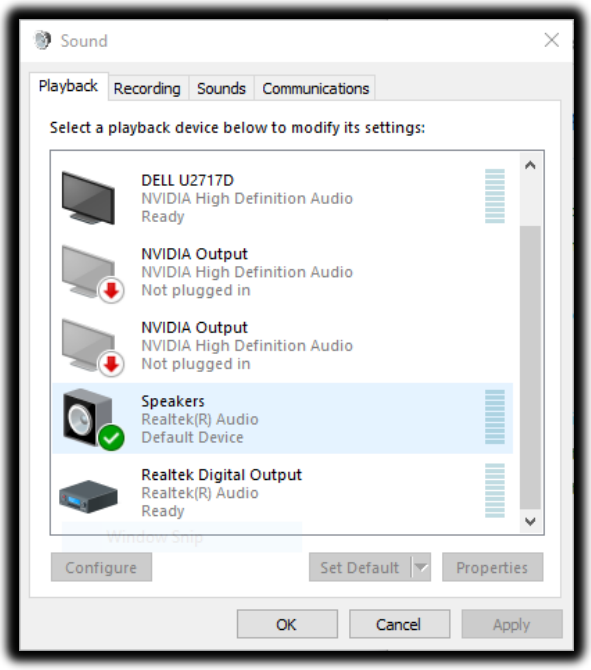


Movies & TV: Play videos, purchase/rent content, supports subtitles & casting.



Sound Settings: Adjust volumes, configure devices, apply enhancements, set defaults.

- **Recording & Editing:**
  - ✓ **Voice Recorder** → Capture audio notes or interviews.
  - ✓ **Photos app** → Basic video editing (trim, add music, effects).
- **Gaming & Streaming:**
  - ✓ **Xbox App & Game Bar** → Capture gameplay, stream games, manage audio.
- **Virtual/Mixed Reality:** Windows Mixed Reality platform for VR/360 content.



3. Core Multimedia Capabilities

Device-Independent Multimedia API

Purpose: Abstract hardware details so apps work across different devices.

Supported hardware:

- **Waveform Audio Devices (Sound Cards)**
  - ✓ Convert analog ↔ digital audio, store in .WAV files, play through speakers.
- **MIDI Devices**
  - ✓ Musical Instrument Digital Interface. Produce notes via keyboards or synthesizers.
- **CD-ROM Drives (CD Audio)**
  - ✓ Play standard audio CDs directly.
- **Video for Windows (AVI)**
  - ✓ Software playback for AVI files, can use hardware acceleration.
- **ActiveMovie Control**
  - ✓ Expand playback to QuickTime, MPEG; hardware acceleration supported.
- **Laserdisc Players & VISCA Video Cassettes**
  - ✓ Controlled via serial interface for PC-driven playback.

Key Concepts

- **Device Abstraction:** Write once, works on any hardware. The API hides device-specific details.
- **Hardware Mixing:** Combine multiple audio sources (wave, MIDI, CD) in **Volume Control**, adjust relative volumes.
- **Hardware Acceleration:** Video boards speed up movie rendering; smoother playback.
- **Serial Interface Control:** Control devices like laserdiscs via PC commands.

4. Evolution & Importance

- **1991 → Now:** Multimedia moved from optional extensions to core part of Windows.
- **Standardization:** Sound cards, CD-ROMs, and video boards became ubiquitous.
- **User Experience:** Multimedia enriches interfaces—beyond text/numbers to **immersive, interactive computing**.

5. WinAPI Takeaways

- Windows provides a **consistent API** for audio, MIDI, CD audio, and video, making **cross-hardware apps possible**.
- Supports both **hardware-accelerated** and **software-based** multimedia.
- Modern Windows (10+) combines **playback, recording, editing, gaming, VR/AR** under a unified multimedia strategy.
- Multimedia isn't just a "feature"—it's a **platform-level layer** that transforms how users interact with their computers.

6. Summary

- **Windows Multimedia = Standard Today:** CD, audio, MIDI, AVI, VR.
- **APIs abstract hardware:** Apps don't need to know exact sound card or video board.
- **User impact:** Windows went from number crunching to entertainment, productivity, and immersive experiences.
- **WinAPI lesson:** Multimedia APIs are a perfect example of **device abstraction, hardware acceleration, and modular hardware support**.

MULTIMEDIA API DESIGN IN WINDOWS (PART 2)

1. Strategic API Design – The Dual-Layer Approach

Windows organizes its multimedia APIs in **two layers**, each with a clear purpose:

Low-Level Interfaces

- **Purpose:** Direct, fine-grained control over hardware.
- **Pros:** Maximum flexibility, precision, and optimization.
- **Cons:** More complex, requires deeper programming knowledge.
- **Examples & Use Cases:**
  - ✓ **Waveform Audio:** waveIn & waveOut → record/playback digital audio (voices, music, sound effects).
  - ✓ **MIDI:** midiIn, midiOut, midiStream → control synthesizers, keyboards, and music sequencing.
  - ✓ **Timing:** time functions → high-resolution timers for syncing audio/video and MIDI events.

High-Level Interfaces

- **Purpose:** Simplify development for common tasks.
- **Pros:** Faster to develop, easier to read/maintain code.
- **Cons:** Less control, limited fine-tuning.
- **Key Example: MCI (Media Control Interface)**
    - ✓ **String-Based Commands:** Control multiple devices (audio, video, CD-ROM) via simple textual commands.
    - ✓ **Rapid Prototyping:** Perfect for experiments or scripting small multimedia projects.
    - ✓ **Device Coverage:** Audio, video, optical media—basically a unified interface for most consumer multimedia.

2. Beyond the Core – Expanding Possibilities

DirectX API

- **Purpose:** Hardware-accelerated multimedia for games and graphics-intensive apps.
- **Features:**
    - ✓ 3D graphics rendering
    - ✓ Advanced audio processing
    - ✓ Robust input support (controllers, VR devices)
- **Usage:** While not deep in this chapter, DirectX is the **go-to for performance-heavy multimedia apps**, especially games.

Convenient Utilities

- **MessageBeep:** Quick sound alerts for UI feedback.
- **PlaySound:** Simplifies playing sound effects or music in apps.
- **Use Case:** Great for adding **immediate audio feedback** without dealing with complex APIs.

3. Key Considerations When Choosing APIs

Project Requirements:

- Real-time audio? Go low-level.
- Simple playback? High-level MCI is enough.

Developer Expertise:

- Low-level APIs require understanding buffers, timing, and device interfaces.
- High-level APIs work for most general apps with standard functionality.

Target Hardware:

- Know your sound cards, MIDI devices, and video capabilities.
- Some APIs (like DirectX) leverage advanced hardware acceleration, while others are software-based.

4. TLDR

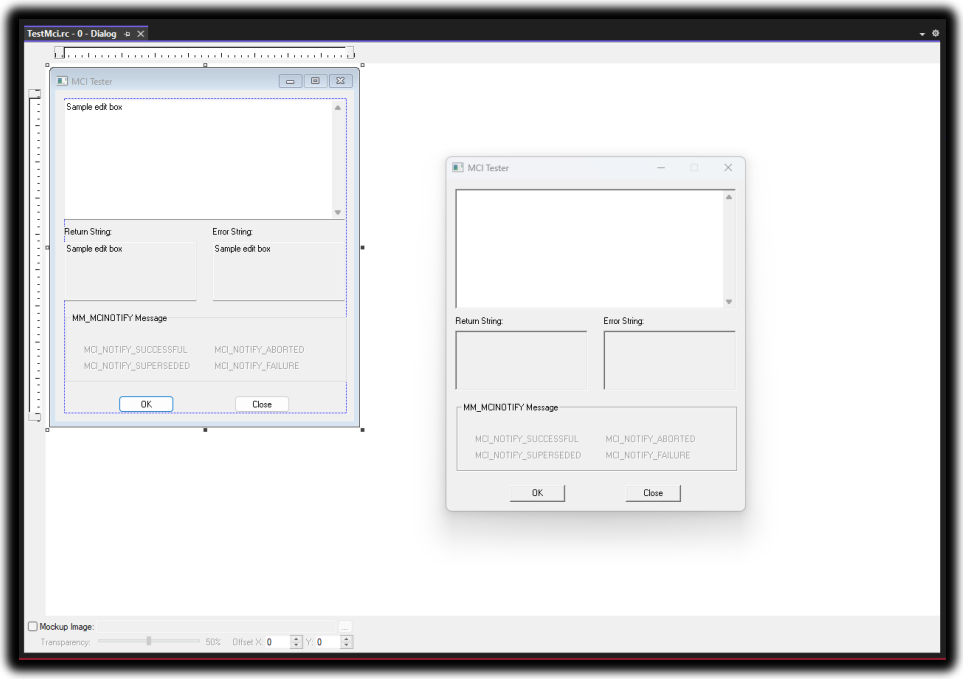
Windows gives **two tiers** of multimedia APIs:

- **Low-Level:** Maximum control & performance (wave, MIDI, time).
- **High-Level:** Fast, easy, versatile (MCI, PlaySound, MessageBeep).

DirectX: Extra power for graphics/audio-heavy apps.

Choice matters: Base it on **project needs, dev skill, and hardware**.

TESTMCI PROGRAM



TESTMCI program in action.mp4

Chapter 22 – TESTMCI Program & MCI Commands 🎶 (Part 3)

1. What TESTMCI Really Does

- It's an **interactive playground for MCI commands**.
- You type in commands, hit Enter/OK, and Windows runs them via **mciSendString**.
- Responses go to the **Return String** box, and errors are nicely translated with **mciGetErrorText**.
- Multiple selected lines? Each one runs in sequence—like mini scripts.

TLDR: It's a **live MCI console** with GUI feedback. Think of it as a "CD player + MCI debugger in one window."

2. Core Functions

mciSendString

- Sends a textual MCI command to the system.
- Examples:
  - ✓ open cdaudio → open the CD drive
  - ✓ play cdaudio → start playback
  - ✓ status cdaudio length → get total CD length

mciGetErrorText

- Converts numeric MCI error codes into readable text.
- Displays errors in the **Error String** section so you know what went wrong.

3. CD Audio Control

- Control a real CD drive with commands like:
  - ✓ pause cdaudio, stop cdaudio → simple playback controls
  - ✓ play cdaudio from 2:30 to 3:00 → play specific time ranges
- Can query: total tracks, track lengths, and time formats (msf, tmsf).

4. Timing & Options

- **wait:** command blocks until it finishes.
- **notify:** sends an MM_MCINOTIFY message when done.
- **break:** safety net if wait is stuck too long.
- **Scripting:** Select multiple lines → batch execution → mini automation.

Pro Tip: Combining wait + notify is possible, but mostly unnecessary. One is usually enough.

5. Potential Applications

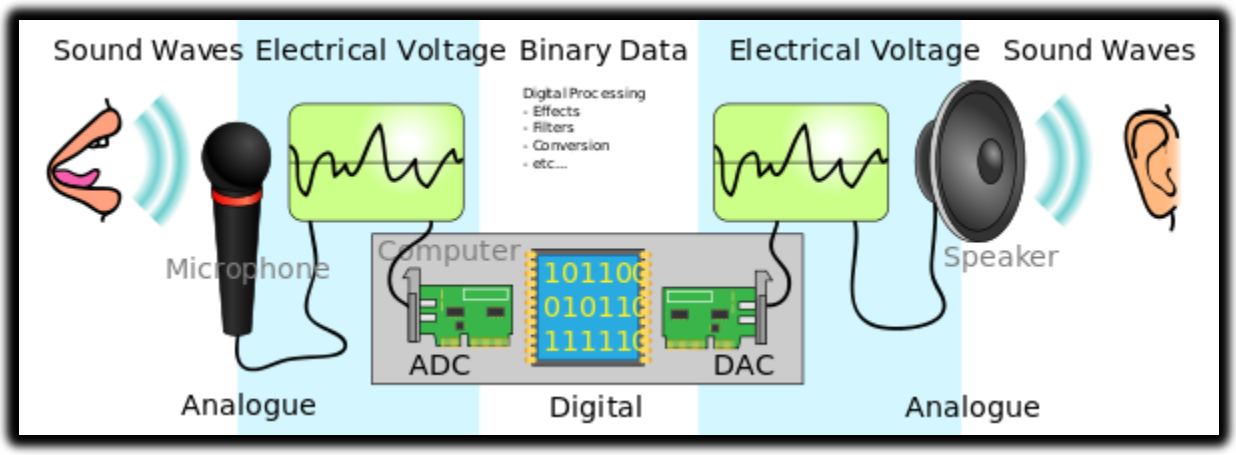
- Build a **mini-CD player UI**: display track info, play/pause, show a timer.
- **Sync graphics with music:** for instruction software or small visualizers.
- Automation of repeated tasks with **MCI scripts**.

6. Bottom Line

TESTMCI is a **hands-on MCI lab**.

- It **teaches by doing**, not just reading.
- You can experiment, batch commands, see errors in real time.
- Overexplaining the GUI and buffers is kinda moot—the **magic is in typing commands and watching Windows respond**.

Chapter 22 – Waveform Audio in Windows 🎵 (Part 4)



1. Waveform Audio Basics

- **Waveform audio = foundation of Windows sound**.
- Converts **analog vibrations → digital samples** → stored as .WAV files.
- Enables **recording, processing, playback**.

Real-world analogy: Microphone captures air pressure changes → Windows digitizes them → you can play them back or manipulate them.

2. Sound 101

- **Sound = vibration** perceived as air pressure changes hitting eardrums.
- **Analog storage:** tapes, records → vibrations stored as magnetic or physical patterns.
- **Digital storage:** digitized waveforms in .WAV files.

Key Parameters:

AUDIO SIGNAL PHYSICS VS. PERCEPTION

Wave Parameter	*Physical Meaning*	Human Perception
Amplitude	*Peak height of the waveform*	Loudness
Frequency	*Cycles per second (Hz)*	Pitch
Complexity	*Harmonic content / Waveshape*	Timbre

- Humans hear roughly **20 Hz – 20 kHz**, but higher frequencies fade with age.
- Human frequency perception is **logarithmic**, not linear → doubling frequency = octave.
- Piano spans ~7 octaves (27.5 Hz – 4186 Hz).

3. Complexity of Real Sounds

- **Pure sine waves = boring sounds**.
- Real sounds = mix of **multiple harmonics** (Fourier series).
  - ✓ Fundamental frequency = base pitch.
  - ✓ Harmonics = integer multiples of fundamental → define **timbre**.
- **Timbre = why a piano sounds different from a trumpet**.
  - ✓ Requires at least 1 fundamental + 7 harmonics.
  - ✓ Harmonics intensity varies across notes and instruments.

Attack & Dynamic Harmonics

- First part of a note (attack) = crucial for recognizing timbre.
- Harmonics evolve dynamically → making **synthesis non-trivial**.

4. Digital Audio Representation

- Advancements in **digital storage** allow **direct digital capture**.
- Instead of reconstructing sound from sine waves, we **store rich, complex sounds** as-is.
- Enables:
    - ✓ Accurate instrument reproduction
    - ✓ Efficient playback & editing
    - ✓ Complex audio applications in Windows

5. Windows Waveform Audio API

- Provides functions to:
    - ✓ **Capture audio** (from mic or line-in)
    - ✓ **Store digitally** (memory or disk)
    - ✓ **Playback** through speakers or other outputs
- Forms the backbone of audio apps in Windows: **voice recorders, games, music software, and more**.

TLDR: Waveform audio in Windows isn't just "play .WAV files." It's the **digital bridge between real-world sound physics and software manipulation**.

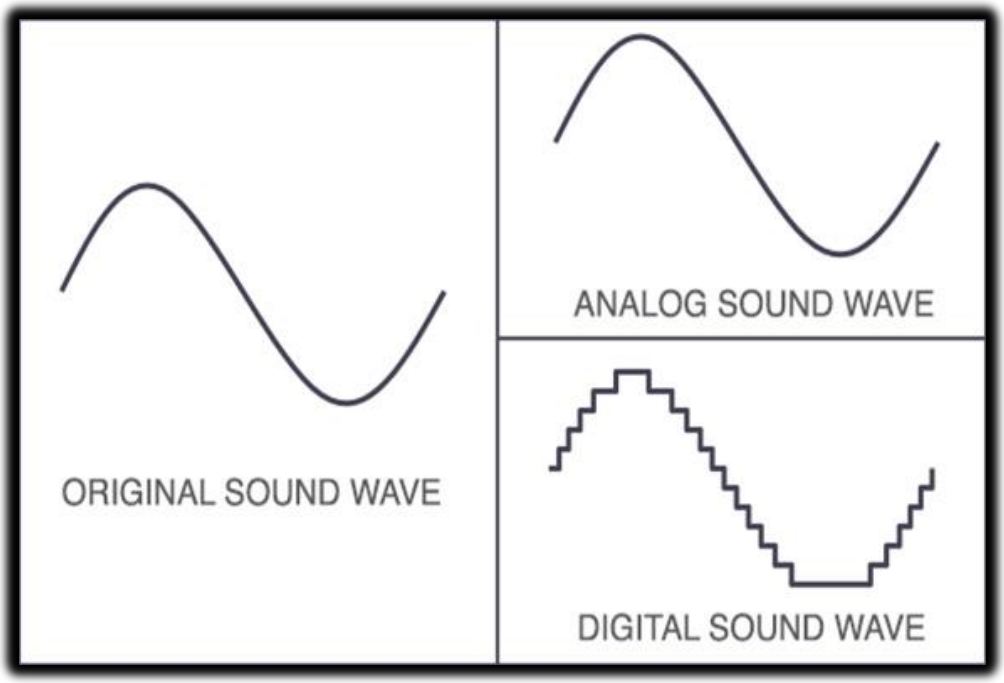
From sine waves → harmonics → timbre → digital storage, understanding this makes you **ready to leverage the Waveform API efficiently**.

Chapter 22 – Pulse Code Modulation (PCM) & Sampling

1. PCM: Bridging Analog Sound & Digital Computers

- **Analog sound** = smooth, continuous waves.
- **Digital computers** = discrete numbers only.
- **PCM (Pulse Code Modulation)** = method for converting analog waves → digital numbers → back to sound.
- **Steps:**
    1. **Sampling:** Take snapshots of the sound wave at regular intervals.
    2. **Quantization:** Convert snapshots into numbers with a set bit depth.
    3. **Digital Storage:** Store as .WAV or memory buffer.
    4. **Playback:** Use DACs + smoothing filters to reconstruct a clean analog waveform.

Analogy: Think of a sound wave as a rolling hill. Sampling = taking photos along the hill; quantization = marking the height of each photo in numbers; DAC + filter = sculpting a smooth hill back from the numbers.

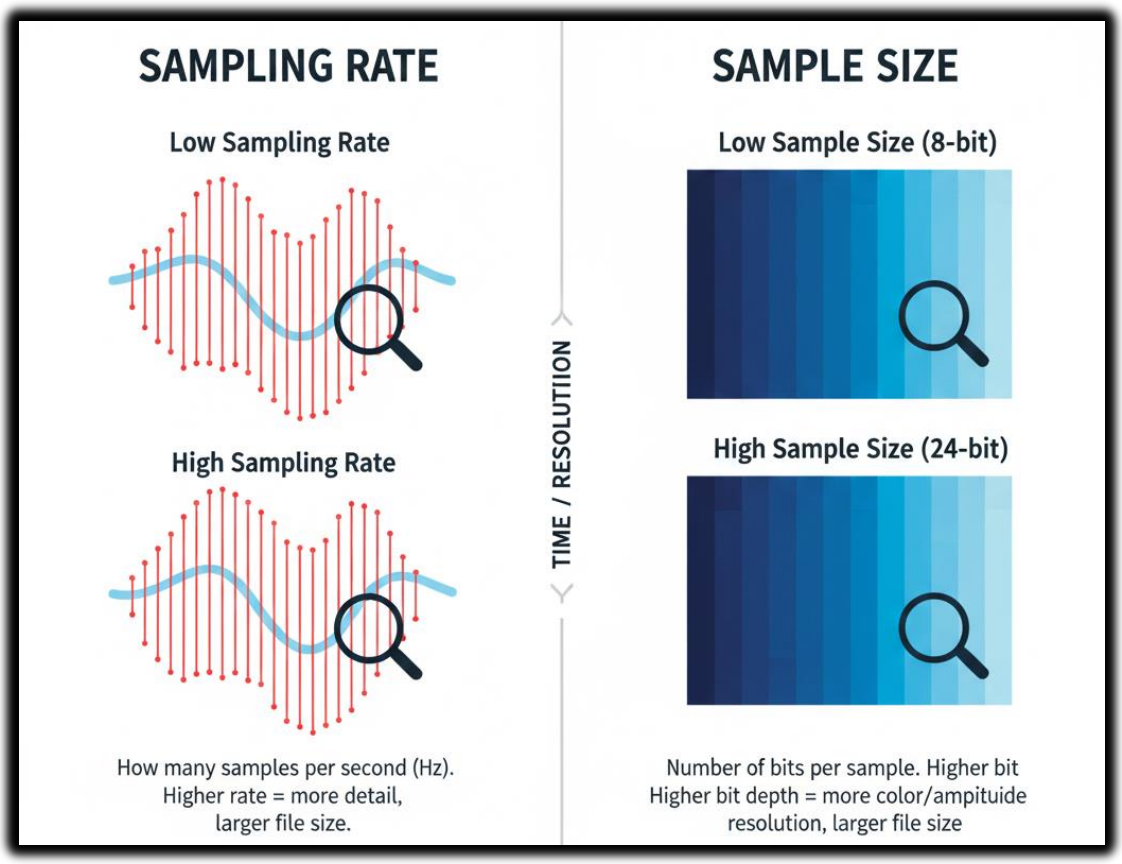


2. Sampling Rate & Sample Size

- **Sampling Rate (Hz):** How often you "take a snapshot" of the waveform per second.
  - ✓ **CD-quality:** 44.1 kHz → captures full human hearing range (~20 Hz–20 kHz).
  - ✓ Lower rates (22.05 kHz, 11.025 kHz, 8 kHz) = smaller file sizes, less detail (good for voice or telephony).
  - ✓ **Nyquist Rule:** Must be ≥ 2 × highest frequency to avoid distortion.
  - ✓ **Aliasing:** Occurs if sampling too slow → creates fake, unwanted frequencies. Prevented with **low-pass filters**.
- **Sample Size (bits):** How precise each snapshot is.
  - ✓ Larger = more detail (like high-res photos of sound).
  - ✓ Smaller = less detail, noisier approximation.

Teen-Friendly Tip:

- Sampling rate = **how often you take pictures of the sound**.
- Sample size = **how many pixels each picture has**.

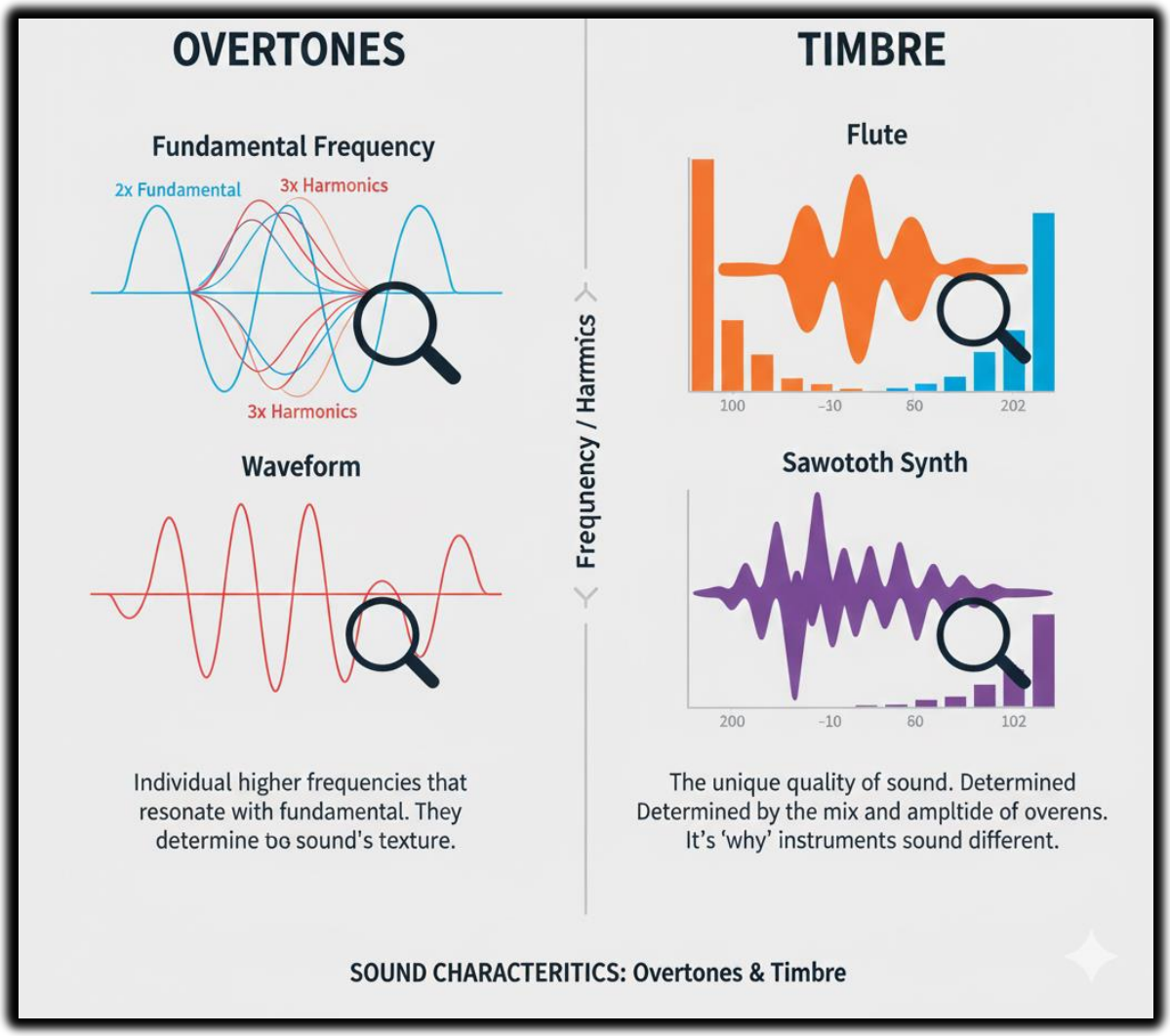


3. Overtones & Timbre

- Real sounds ≠ single sine wave; instruments produce **harmonics (overtones)**.
- **Fundamental frequency:** main pitch.
- **Overtones:** integer multiples of fundamental → create richness & character (timbre).
- Low sampling rates can **miss overtones**, making sound thin or flat.

Analogy:

- Fundamental = main ingredient in a recipe.
- Overtones = spices/flavors.
- Capture both → tasty/full music; capture only fundamental → bland music.



4. Key Takeaways for PCM & Windows Audio

- Sampling rate & sample size = **digital audio quality drivers**.
- **Nyquist frequency** sets minimum sample rate to avoid aliasing.
- **Low-pass filters** prevent digital artifacts.
- Overtones are essential for **true musical richness**.
- Choosing the right balance = **efficient, high-quality audio**.

💡 **Pro tip for coding in Windows:** When using waveIn* or waveOut* APIs, you'll often configure **sampling rate, bits per sample, and channels**. Understanding PCM is critical because these parameters determine whether your recording/playback is crisp, faithful, or just noise.

SAMPLE SIZE: CAPTURING SOUND'S DYNAMIC RANGE

Measuring Sound Detail

Sample size (measured in bits) controls how much detail we can capture in sound. It decides how accurately a digital system can represent quiet sounds and loud sounds.

In short, it defines the **dynamic range** — the distance between the softest and loudest audio we can record without distortion.

More Bits, More Detail

Each extra bit doubles the number of possible volume levels. This gives the system more precision when recording sound.

Think of it like painting: more bits mean more shades of color, so the final picture sounds smoother and more realistic.

Decibels: Measuring Loudness

Why Decibels Are Used

Human hearing does not work in a straight line. We notice small changes in quiet sounds more easily than changes in loud sounds. Because of this, sound intensity is measured on a **logarithmic scale**, called decibels (dB).

Bels and Decibels

The bel is named after Alexander Graham Bell.

- One bel means the sound intensity increased ten times.
- A decibel is one-tenth of a bel and represents a very small but noticeable change in loudness (about 1.26 times louder).

This scale matches how our ears actually hear sound.

$$DR_{dB} = 20 \cdot \log_{10}\left(\frac{A_{\max}}{A_{\min}}\right)$$

Where:

- DR_{dB} is the dynamic range in decibels.
- $A_{\max}$ is the maximum sound amplitude.
- $A_{\min}$ is the minimum sound amplitude.

$$DR_{dB} = 20 \cdot \log_{10}\left(2^{n-1}\right)$$

Where:

- DR_{dB} is the dynamic range in decibels.
- n is the number of bits in the sample size.

Calculating Dynamic Range

The dynamic range between two sounds is measured in decibels using this formula:

Dynamic Range (dB) = 20 × $\log_{10}$ (Maximum Amplitude / Minimum Amplitude)

In digital audio systems, dynamic range depends directly on sample size (number of bits). More bits allow a bigger difference between the quietest and loudest sounds. A common rule is:

Dynamic Range (dB) ≈ 6 × Number of Bits

So:

- 8-bit audio ≈ 48 dB
- 16-bit audio (CD quality) ≈ 96 dB
- 24-bit audio ≈ 144 dB

Storage Space for Uncompressed Audio

To calculate how much storage uncompressed audio needs, we use this formula:

Storage = Sample Rate × Bit Depth × Number of Channels × Duration

For example:
One hour of CD-quality audio:

- 44,100 samples per second
- 16 bits per sample
- Stereo (2 channels)

This requires about **635 megabytes**, which is roughly the capacity of a standard audio CD.

```
Storage Space=Duration (seconds)×Sampling Rate×Sample Size (bytes)×Number of Channels
```

Final Takeaway

Higher bit depth means:

- Better dynamic range
- More accurate sound
- Larger file sizes

So, the better the sound quality, the more storage space we need to preserve all those musical details 🎵

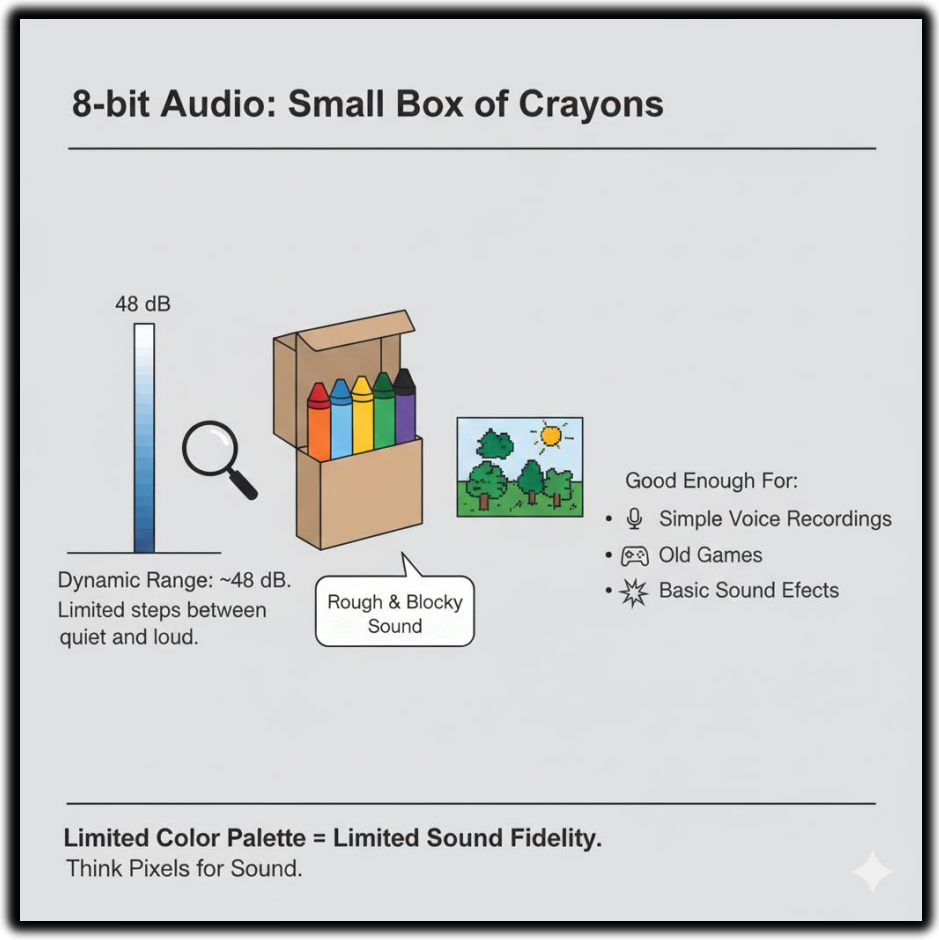
Common Sample Sizes and What They Mean

8-bit Audio

8-bit audio has a dynamic range of about **48 dB**.
This means it can represent only a small range between quiet and loud sounds.

Think of it like a small box of crayons.
You can draw the picture, but the sound is rough and blocky.
It's good enough for:

- Simple voice recordings
- Old games
- Basic sound effects



16-bit Audio

16-bit audio has a dynamic range of about **96 dB** and is the standard for CDs and most consumer audio.

This is like a full box of crayons.
You get smooth sound with enough detail to handle:

- Quiet whispers
- Loud music
- Strong drum hits

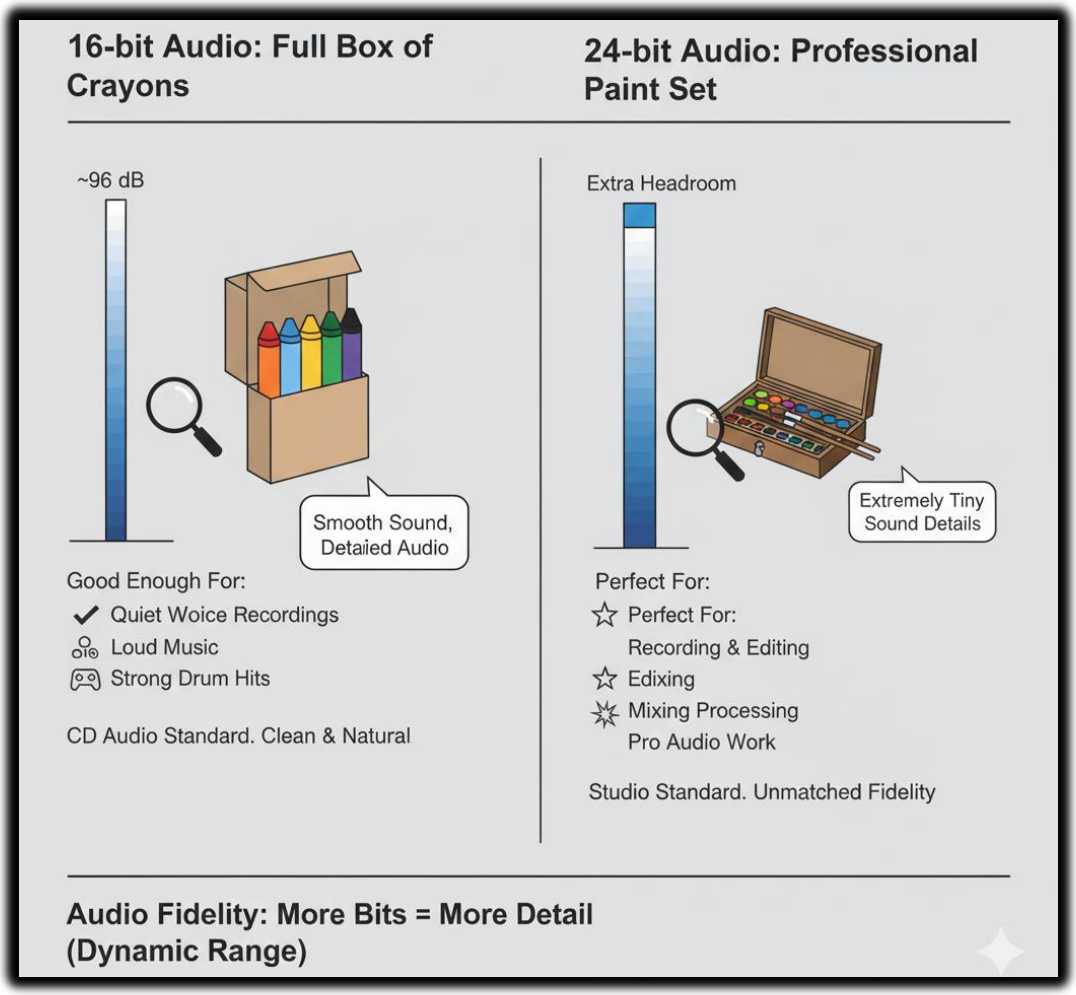
For most listening, 16-bit audio already sounds very clean and natural.

24-bit Audio

24-bit audio offers a huge dynamic range of about **144 dB** and is used in professional recording studios.

This is like having professional paint instead of crayons.
It captures extremely tiny sound details and leaves extra “headroom” so engineers can edit, mix, and process audio without distortion.

Most people can’t hear the full range, but it’s perfect for recording and editing before the final version is made.



Storage and How Audio Data Is Stored

How Samples Are Stored in Windows

Windows commonly supports:

- **8-bit samples** stored as *unsigned bytes*
- **16-bit samples** stored as *signed integers*

You don't need to memorize this, just understand the idea:
different bit sizes use different number formats to store sound levels.

How Silence Looks in Data

Silence is not always stored as "nothing."

- In **8-bit audio**, silence is stored as the value **0x80** (the middle of the range).
- In **16-bit audio**, silence is stored as **all zeros**.

Think of silence as the "center line" of a waveform.
No movement above or below that line means no sound.

How Much Space Uncompressed Audio Needs

Several things decide how big an audio file will be:

- How long the sound is
- How many samples are taken each second (sample rate)
- How many bits each sample uses (bit depth)
- How many channels there are (mono or stereo)

Simple idea:
More samples + more bits + more channels = bigger files.

Real-World Example

One hour of CD-quality audio:

- 44,100 samples per second
- 16 bits per sample
- Stereo (2 channels)

This needs about **635 megabytes** of storage.

Think of it like taking photos:
Higher resolution photos look better, but they take up more space.
Audio works the same way.

Big Picture Takeaway

- Higher bit depth = better sound detail
- Better sound detail = more storage space
- Choose what fits your needs: quality vs file size

Generating Sine Waves in Windows

1. Core Concept

- Digital audio uses **PCM (Pulse Code Modulation)**.
- The sin function from math.h generates sine wave values.
- **Phase angle** ensures smooth, continuous waves—never let it jump suddenly, or you get clicks/glitches.

2. Workflow of the Sine Wave Generator

1. **Set sample rate:** e.g., 11,025 Hz.
2. **Set frequency:** user-controlled (iFreq).
3. **Calculate samples per cycle:** samplesPerCycle = sampleRate / frequency.
4. **Fill buffer:** for each sample, compute amplitude = sin(phaseAngle) * maxAmplitude.
5. **Increment phase angle:** phaseAngle += 2 * PI * frequency / sampleRate.
6. **Wrap phase angle:** subtract 2*PI if it exceeds 2π.
7. **Double-buffer playback:** pBuffer1 & pBuffer2 keep audio continuous without gaps.

3. Windows Waveform Audio API

- waveOutOpen: Opens audio device, specifying device ID (WAVE_MAPPER) and waveform format (WAVEFORMATEX).
- waveOutPrepareHeader: Prepares buffers so they won't be swapped to disk.
- waveOutWrite: Sends buffer to hardware for playback.
- waveOutReset: Stops playback immediately.
- **Double-buffering & MM_WOM_DONE messages** = seamless audio streaming.

4. User Interface

- **Scroll bar:** Adjust frequency in real-time (20 Hz → 5000 Hz).
- **Text box:** Displays frequency, allows manual input.
- **On/Off button:** Starts/stops playback.

Interactive magic: move scroll bar → instantly hear frequency change.

5. Waveform Format & Buffers

- WAVEFORMATEX defines: sample rate, bits per sample, channels, block alignment.
- WAVEHDR defines: buffer pointer, buffer length, loops, flags.
- Use **double-buffering** to continuously feed the audio device.

Tip: dwLoops = 1 → repeat once per buffer, handled by the API via MM_WOM_DONE.

6. Messages & Cleanup

- **MM_WOM_OPEN:** Device ready → fill buffers and start playback.
- **MM_WOM_DONE:** Hardware finished buffer → refill & write next buffer.
- **MM_WOM_CLOSE:** Device closed → free buffers, reset UI.
- **WM_SYSCOMMAND (SC_CLOSE):** Graceful shutdown if user closes window.

Result: smooth, continuous, adjustable sine wave playback with interactive control.

7. Key Takeaways

- Double-buffering + phase management = **no audio glitches**.
- Real-time frequency adjustment = **interactive experience**.
- Proper WAVEFORMATEX and WAVEHDR setup = **stable playback**.
- Messages (MM_WOM_*) = **audio device feedback loop**.

The cool part here is that this approach is **literally everything you need to build a mini synthesizer or a test tone generator** in Windows.

Once you fully grasp double-buffering + MM_WOM_DONE + phase continuity, you can make **any waveform, not just sine waves**—triangles, squares, even complex harmonics. 🎛️

SinWave program.mp4

RECORD.C – Windows Audio Recording & Playback 🎤🎶

1. Program Initialization

- **Headers included:** windows.h + resource.h.
- **Memory allocated:**
  - ✓ pWaveHdr1, pWaveHdr2 → wave headers for audio buffers.
  - ✓ pSaveBuffer → stores recorded audio.
- **Platform check:** Windows NT? Then show main dialog.
- **WinMain:** Entry point, sets up everything.

2. Utility: ReverseMemory

- Reverses audio data in a buffer → enables **reverse playback**.
- Swaps bytes from start ↔ end of buffer.
- Simple but essential for creative playback features.

3. Dialog Procedure (DlgProc)

Handles **all UI + audio logic** via messages (WM_COMMAND, MM_WIM_DATA, MM_WOM_DONE, etc.).

4. Recording Flow

Trigger: Press “Record Begin” (IDC_RECORD_BEG)

Steps:

1. **Allocate input buffers:** pBuffer1 and pBuffer2 (size INP_BUFFER_SIZE).
2. **Open audio input:** waveInOpen with WAVEFORMATEX for format, hwnd for callbacks.
3. **Prepare wave headers:** waveInPrepareHeader.
4. **Start recording:** waveInAddBuffer + waveInStart.

Data Handling (MM_WIM_DATA):

- Reallocate pSaveBuffer via realloc to store incoming data.
- Copy current buffer data into pSaveBuffer.
- If ending (bEnding=TRUE) → waveInClose; else, queue next buffer.

End Recording (IDC_RECORD_END):

- Set bEnding = TRUE.
- waveInReset stops recording, flushes last buffer → triggers MM_WIM_DATA.
- waveInClose frees resources, enables/disables buttons.

5. Playback Flow

Trigger: Press “Play Begin” (IDC_PLAY_BEG)

Steps:

1. Initialize WAVEFORMATEX with sample rate, channels, bits/sample.
2. Open audio output device: waveOutOpen.
3. Prepare header with pSaveBuffer: waveOutPrepareHeader.
4. Start playback: waveOutWrite.

Data Handling (MM_WOM_DONE):

- Playback finished → unprepare header + waveOutClose.
- If bReverse=TRUE → call ReverseMemory for next reverse playback.

End Playback: System closes properly via MM_WOM_CLOSE.

6. Playback Enhancements

- **Pause/Resume:** waveOutPause / waveOutRestart.
- **Reverse:** ReverseMemory.
- **Repeat:** Set dwLoops and dwFlags in WAVEHDR.
- **Speedup:** Adjust nSamplesPerSec and nAvgBytesPerSec in WAVEFORMATEX → doubles playback speed.

7. Memory & Resource Management

- Memory allocated for:
  - ✓ Input buffers (pBuffer1, pBuffer2)
  - ✓ Save buffer (pSaveBuffer) → dynamically resized during recording.
- Freed during:
  - ✓ MM_WIM_CLOSE, MM_WOM_CLOSE, program exit.
- Efficient handling ensures **no leaks** during extended recording/playback sessions.

8. Key Insights

- **Double-buffering** ensures smooth continuous audio capture and playback.
- **MM_WIM_DATA / MM_WOM_DONE** = backbone of async handling for audio devices.
- **WAVEFORMATEX + WAVEHDR** = central structures for all waveform audio operations.
- **Reverse / Repeat / Speedup / Pause / Resume** = shows power & flexibility of the Windows Waveform API.

💡 TLDR:

RECORD.C is basically a **mini DAW in a dialog box**. It captures audio, allows playback, pause, reverse, repeat, and speed adjustment—all via low-level waveform API. If you nail MM_WIM_DATA, MM_WOM_DONE, double-buffering, and ReverseMemory, you've got full control of real-time digital audio in Windows.

Windows Audio Programming – RECORD1 → RECORD3 🎧

1. Low-Level Waveform API (RECORD1)

Mechanism:

- Direct hardware access via waveInOpen, waveOutOpen, waveInAddBuffer, waveOutWrite.
- Double-buffering (pBuffer1/pBuffer2) + dynamic pSaveBuffer for recording.
- Handles audio at byte level → supports effects like **reverse, repeat, speedup, pause/resume**.

Message Handling:

- WM_COMMAND → buttons: record, play, pause, stop.
- MM_WIM_DATA → recorded buffer available.
- MM_WOM_DONE → playback finished.

Pros: Fine-grained control → special effects, low-latency recording/playback.

Cons: Complex: memory management, buffer handling, messages, headers.

2. Message-Based MCI (RECORD2)

Mechanism:

- File-based audio: records to .wav files.
- Controls device via mciSendCommand(wDeviceID, command, flags, params).
- Key commands: MCI_OPEN, MCI_RECORD, MCI_STOP, MCI_SAVE, MCI_PLAY, MCI_PAUSE, MCI_CLOSE.
- MM_MCINOTIFY messages used to track completion: MCI_NOTIFY_SUCCESSFUL or MCI_NOTIFY_ABORTED.

Workflow:

- Record Begin → MCI_OPEN + MCI_RECORD.
- Record End → MCI_STOP → MCI_SAVE → MCI_CLOSE.
- Playback → MCI_OPEN file → MCI_PLAY.
- Pause/Resume → MCI_PAUSE / MCI_PLAY.

Pros:

- Simpler: no manual buffer management.
- Automatic file storage and retrieval.
- Built-in notifications for operation completion.

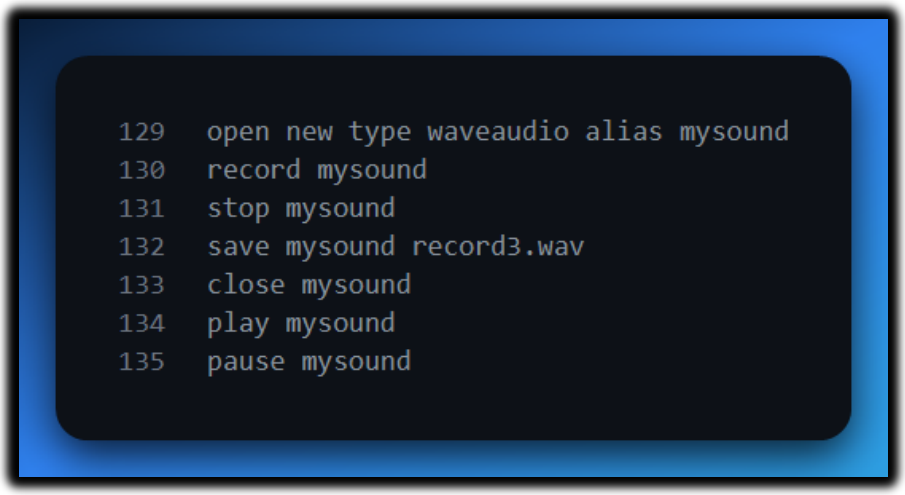
Cons:

- File-based → no real-time effects (reverse, speedup).
- Slightly slower than low-level API for intensive processing.

3. MCI String Approach (RECORD3)

Mechanism:

- Uses **text commands** (mciSendString / mciExecute) instead of message structs.
- Examples:



```
129  open new type waveaudio alias mysound
130  record mysound
131  stop mysound
132  save mysound record3.wav
133  close mysound
134  play mysound
135  pause mysound
```

- Alias allows referencing device by name rather than ID.

Workflow:

- Record → delete existing file → open → record.
- Stop → stop → save → close.
- Play → open file → play.
- Pause/Resume → pause / play.

Differences from RECORD2:

- No MM_MCINOTIFY → program doesn't auto-detect completion.
- Buttons must be manually clicked for end-of-playback/recording actions.
- Encapsulated error handling via mciExecute (returns Boolean).

Pros:

- Very concise, readable code.
- Easier for scripting, command-line style, and cross-Windows versions.

Cons:

- Less control over device (no real-time processing).
- No automatic state management → user must manually end operations.
- Limited to MCI-supported formats and features.

4. Key Differences Across RECORD1 → RECORD3

Win32 Multimedia: Audio Recording Implementations

FEATURE	RECORD1 (WAVEFORM)	RECORD2 (MCI MESSAGE)	RECORD3 (MCI STRING)
API Architecture	Low-level Waveform API	MCI Message-based	MCI String-based
Data Handling	Memory buffers (Double-buffering)	Disk Files	Disk Files
Special Effects	Reverse, Repeat, Speedup	None	None
Notifications	`MM_WIM_DATA` / `MM_WOM_DONE`	`MM_MCINOTIFY`	None (Manual Polling)
Complexity	**High** (Buffer Mgmt)	Medium	Low
Performance	**High / Real-time**	Moderate	Moderate
File Handling	Optional (Manual I/O)	Required	Required
Error Handling	Manual Error Codes	Manual + `mciGetErrorString`	Wrapped in `mciExecute`
UI Dependency	Manual Button States	Auto via Notifications	Manual update loop

5. Practical Notes / Takeaways

- **Low-Level API** → use for learning, special effects, or real-time audio apps.
- **Message-Based MCI** → good for standard record/play apps with simple automation.
- **String-Based MCI** → easiest to code, readable, cross-Windows, but loses automation and effects.
- **MCI String Examples in real apps:**
  - ✓ Simple audio players (Windows Media Player classic UI).
  - ✓ Voice recorders or note apps.
  - ✓ Lightweight VoIP or chat apps (older Windows platforms).
- **Limitations of MCI String Approach:**
  - ✓ Limited control and effects.
  - ✓ Performance not optimal for live processing.
  - ✓ Windows-only, older tech (not updated for modern audio features).

6. Waveform Audio File Format

Delving into the uncompressed (PCM) .WAV files reveals a specific format, as outlined in Figure 22-6.

Offset (Bytes)	Bytes	Data	Description
0000	4	"RIFF"	File identifier
0004	4	Size of waveform chunk	Size of the waveform chunk (file size minus 8)
0008	4	"WAVE"	Format identifier
000C	4	"fmt "	Format chunk identifier
0010	4	16	Size of format chunk
0014	2	1	Format tag (WAVE_FORMAT_PCM for uncompressed audio)
0016	2	wf.nChannels	Number of audio channels (mono or stereo)
0018	4	wf.nSamplesPerSec	Sample rate (samples per second)
001C	4	wf.nAvgBytesPerSec	Average bytes per second
0020	2	wf.nBlockAlign	Block alignment (bytes per sample)
0022	2	wf.wBitsPerSample	Bits per sample (audio resolution)
0024	4	"data"	Data chunk identifier
0028	4	Size of waveform data	Size of the waveform data
002C	...	Waveform data	Raw audio samples

Understanding the WAV (RIFF) File Structure

A **WAV file** is not just raw sound data.
It is a **well-organized container** that tells your program **how** the sound data is stored and **how to play it correctly**.

Think of a WAV file like a labeled box 

Inside the box are smaller labeled sections. Each section has:

- A **name** (what this part is for)
- A **size** (how much data it contains)
- The **actual data**

These sections are called **chunks**.

I. The RIFF Structure (Big Picture)

WAV files use the **RIFF** (Resource Interchange File Format).

RIFF works like this:

- Every chunk starts with a **4-character name** (ASCII text)
- Followed by a **4-byte size value**
- Then the actual data

This makes the file easy to read and easy to extend.

II. Main Chunks in a WAV File

a) The "RIFF" Chunk (File Header)

This is the **very first thing** in the file.

It tells us:

- "This is a RIFF file"
- How big the rest of the file is
- That the file contains **WAVE** data

So, the start looks like:

- "RIFF" → file type
- File size → how much data follows
- "WAVE" → confirms it's a WAV audio file

b) The "fmt " Chunk (Audio Settings)

This chunk explains **how the sound data is stored**.
Without it, the audio data would be meaningless noise.

Think of this chunk as the instruction manual 📘 for reading the sound.

Important fields inside the fmt chunk:

- **nChannels**
  - ✓ 1 = mono
  - ✓ 2 = stereo
- **nSamplesPerSec**
  - ✓ How many samples are taken each second
  - ✓ Common values: 11025, 22050, 44100
- **wBitsPerSample**
  - ✓ How detailed each sample is (8-bit, 16-bit, etc.)
- **nBlockAlign**
  - ✓ How many bytes one complete sample uses
  - ✓ (all channels combined)
- **nAvgBytesPerSec**
  - ✓ How many bytes are played each second
  - ✓ Used to estimate playback speed and duration

Usually, this chunk is **16 bytes long** for standard PCM audio.

c) The “data” Chunk (The Actual Sound)

This chunk contains the **real audio samples**.

This is the music itself 🎶.

The chunk includes:

- "data" identifier
- Size of the audio data
- The sample values, stored one after another

How samples are stored:

- **8-bit audio**
  - ✓ Mono: 1 byte per sample
  - ✓ Stereo: 2 bytes (left, then right)
- **16-bit audio**
  - ✓ Mono: 2 bytes per sample
  - ✓ Stereo: 4 bytes (left, then right)

In stereo audio, samples always go:
Left sample → Right sample

d) Extra Chunks (Optional but Normal)

WAV files may contain extra chunks like:

- "INFO" (metadata such as title or artist)

These chunks are **not required** to play the sound.

Important Rule 🔑

When reading a WAV file:

- **Ignore chunks you don’t recognize**
- Only process the chunks you need (fmt and data)

This makes your program more robust and future-proof.

III. Why This Matters for Programming

If you want to:

- Load WAV files manually
- Write your own audio player
- Process sound data

You **must understand this structure**.
Otherwise, you won't know:

- How fast to play the samples
- How many channels exist
- How to interpret the raw numbers

IV. Drawback of mciExecute in RECORD3

The mciExecute function is **easy to use**, but it has a big limitation.

The Problem

- The program is **not notified** when playback finishes
- No MM_MCINOTIFY message is sent

What This Means

- The program cannot automatically update button states
- It doesn't know when the sound ends

Result

The user must manually press the **"End"** button to signal completion.

In short: mciExecute is simple, but you lose control and feedback.

V. Final Big Picture Takeaway

- WAV files are **structured containers**, not raw sound
- RIFF chunks make the format flexible and expandable
- The fmt chunk explains *how* to read the sound
- The data chunk contains the actual audio
- Ignoring unknown chunks makes your code safer
- Simpler APIs like mciExecute trade power for convenience

EXPLORING ADDITIVE SYNTHESIS AND TIMBRE IN MUSICAL TONES

When we analyze musical sounds, we usually start with three basic ideas: **pitch**, **volume**, and **timbre**.

Pitch and Volume: The Easy Parts

Pitch is simply how high or low a sound is. It depends on frequency, measured in hertz (Hz).
Humans can hear roughly from **20 Hz to 20,000 Hz**. Musical instruments use only part of this range. For example, a piano spans from about **27.5 Hz** at the lowest note to **4186 Hz** at the highest.

Volume, or loudness, tells us how strong the sound is. It is related to the **amplitude** of the sound wave and is measured in **decibels (dB)**.

These two properties are fairly straightforward. The real mystery begins with **timbre**.

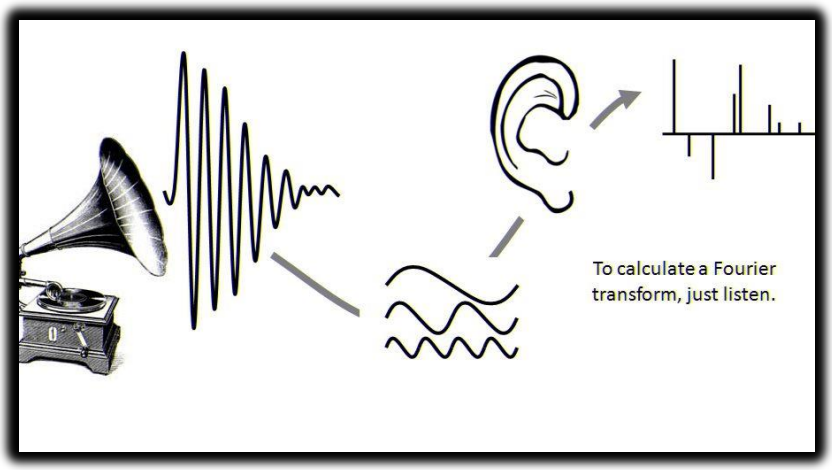
Timbre: Why Instruments Sound Different

Timbre is what lets us tell the difference between a piano, a violin, and a trumpet—even if they all play the *same note* at the *same loudness*.

You can think of timbre as an instrument's **sound fingerprint**.
It is not about pitch or volume, but about the *shape and structure* of the sound wave itself.

Fourier's Big Idea: Sound as Building Blocks

The key to understanding timbre comes from mathematics.
Joseph Fourier showed that **any periodic waveform**, no matter how complex it looks, can be built by adding together **simple sine waves**.



Each of these sine waves has:

- A **frequency**
- An **amplitude**

One sine wave represents the **fundamental frequency**, which sets the pitch of the sound. The rest are called **overtones** or **harmonics**, and their frequencies are whole-number multiples of the fundamental.

For example:

- First overtone (2nd harmonic): 2 × fundamental frequency
- Second overtone (3rd harmonic): 3 × fundamental frequency
- And so on

The **relative strength** (amplitude) of these harmonics is what shapes the waveform and defines the timbre.

Wave Shapes and Harmonics

Different waveforms contain different harmonic patterns.

A **square wave** contains:

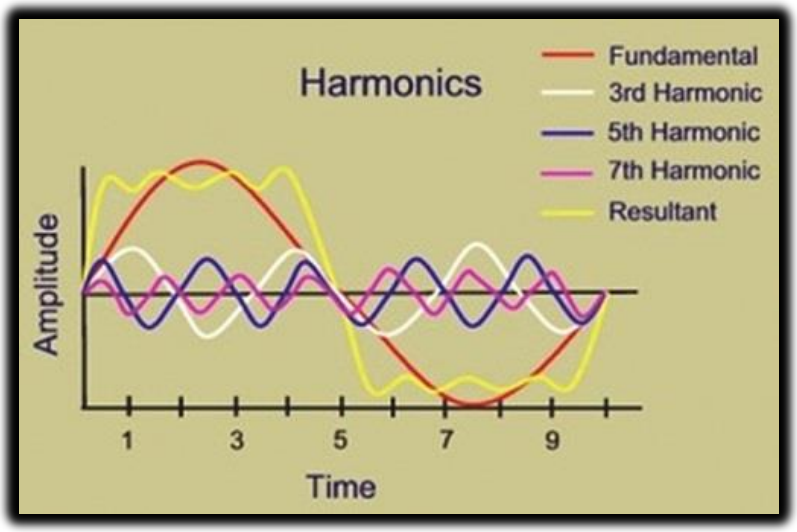
- Only odd harmonics
- Amplitudes that decrease like: 1, 1/3, 1/5, 1/7, ...

A **sawtooth wave** contains:

- All harmonics (odd and even)
- Amplitudes that decrease like: 1, 1/2, 1/3, 1/4, ...

By adding these sine waves together in the right proportions, we can recreate these complex shapes.
This process is called **additive synthesis**.

Additive synthesis helps us see that timbre is not magic—it is the result of how harmonics are mixed together.



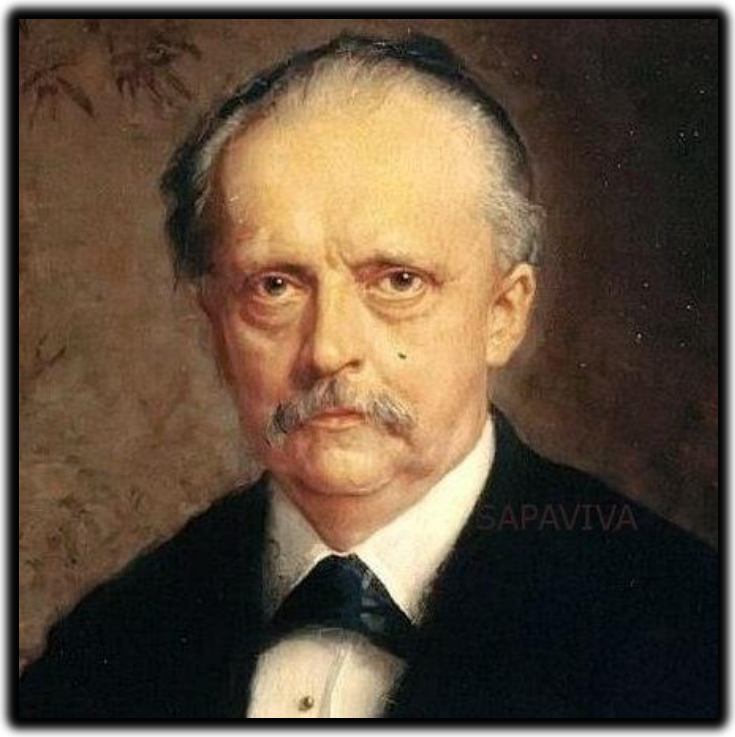
From Science to Music: Helmholtz and Early Theory

In the 19th century, **Hermann Helmholtz** made major contributions to our understanding of sound.
In his book *On the Sensations of Tone* (1885), he suggested that the ear and brain analyze sound by breaking it into sine waves.

According to Helmholtz:

- The ear detects individual frequency components
- The brain uses their relative strengths to determine timbre

This idea was revolutionary, but later research showed that real musical sounds are **more complex** than this simple model.



The Rise of Electronic Music and Analog Synthesizers

A major turning point came in **1968** with Wendy Carlos's album *Switched-On Bach*. This album introduced analog synthesizers, like the **Moog**, to a wide audience.

Analog synthesizers generate basic waveforms:

- Square waves
- Triangle waves
- Sawtooth waves

These simple waves are then shaped to sound more musical.

Envelopes: Shaping a Note Over Time

One important tool in analog synthesis is the **envelope**, which controls how the volume of a note changes.

An envelope usually has three main stages:

- **Attack**: The sound rises quickly from silence
- **Sustain**: The sound stays at a steady level
- **Release**: The sound fades back to silence

This allows a note to behave more like a real instrument instead of a flat, mechanical tone.

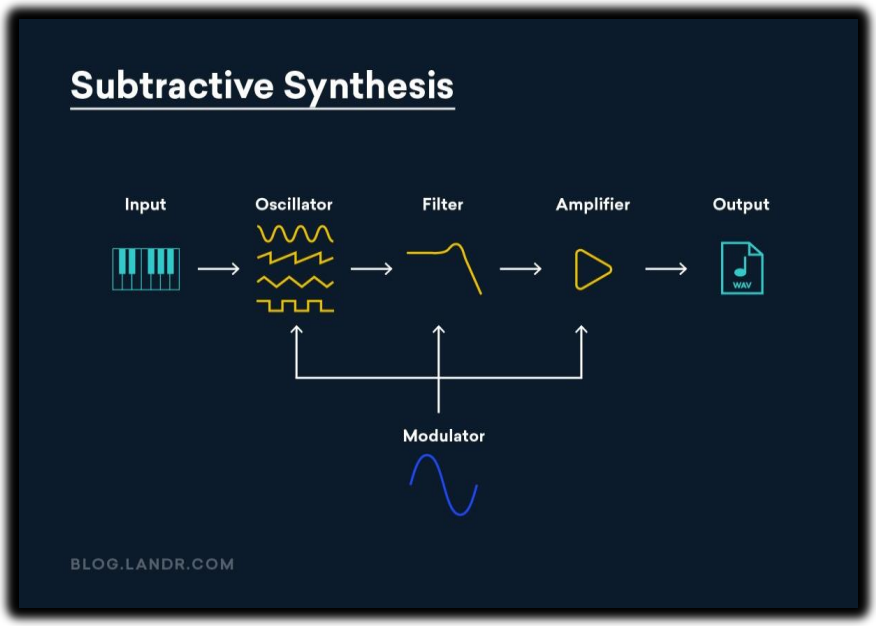
Filters and Subtractive Synthesis

Analog synthesizers also use **filters** to shape sound.

Filters remove or weaken certain harmonics, changing the tone.
The filter's **cutoff frequency** can change over time, often controlled by an envelope.

This approach—starting with a rich waveform and removing parts of it—is called **subtractive synthesis**.

Subtractive synthesis became very popular, but researchers and musicians were still interested in additive synthesis.



The Challenge of Additive Synthesis

Additive synthesis works by:

- Creating many sine waves
- Tuning them to different harmonics
- Controlling each one's amplitude separately

In theory, this is very powerful.
In practice, it was difficult with analog hardware.

A single note might require **8 to 24 sine wave generators**, all perfectly tuned.
Analog oscillators tend to drift in frequency, making this approach unstable and impractical.

Digital Synthesis and Real Musical Sounds

The situation changed with **digital synthesizers** and computer-generated sound.

Digital systems:

- Do not drift in frequency
- Allow precise control over each sine wave

This made additive synthesis much more realistic.

The process typically works like this:

1. Record a real instrument sound
2. Use Fourier analysis to break it into partials
3. Rebuild the sound using multiple sine waves

Why Real Instruments Are More Complex

As researchers analyzed real musical tones, they discovered something important: real instruments are **not perfectly harmonic**.

The frequency components are better called **partials**, not harmonics, because:

- Their frequencies are not exact multiples of the fundamental
- They change over time

This **inharmonicity** is critical.
Perfectly harmonic sounds tend to sound artificial or "electronic."
Small imperfections make the sound feel alive and real.

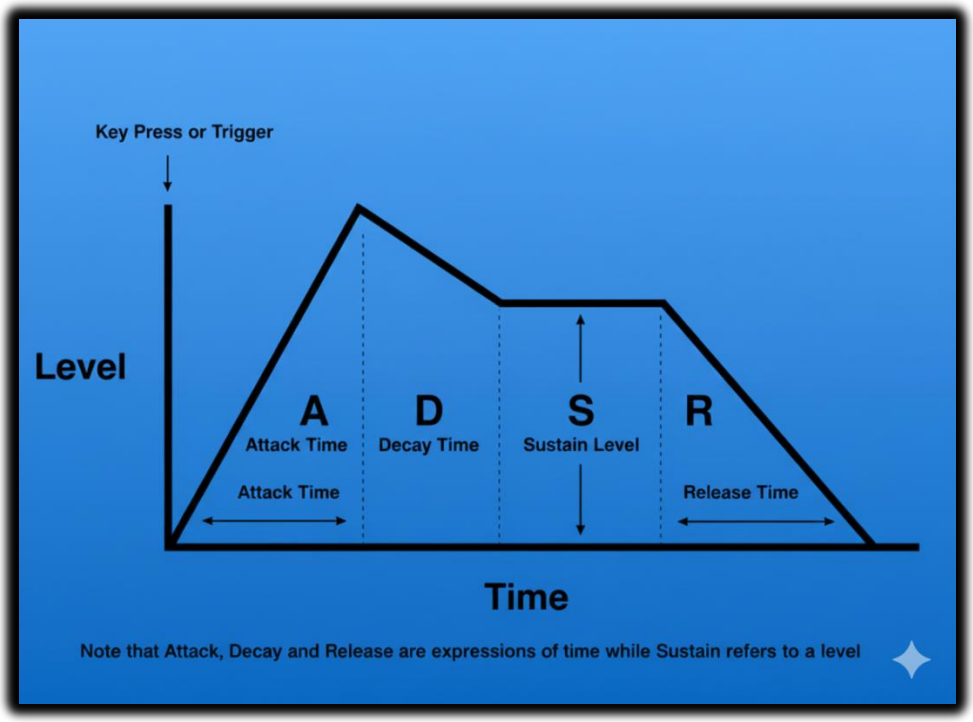
The Importance of the Attack Phase

The most complex part of a musical note is the **attack phase**.

During the attack:

- Frequencies shift
- Amplitudes change rapidly
- Inharmonicity is strongest

Our ears rely heavily on this short moment to identify instruments.
This is why simple synthesis models often fail to sound realistic.



From Research to Real Programs

Early research in the late 1970s, published in journals like *Computer Music Journal*, produced detailed data sets such as the **Lexicon of Analyzed Tones**.

These studies recorded:

- Frequency changes over time
- Amplitude changes for each partial

With support for waveform audio in Windows, it became possible to turn this data into sound.

The **ADDSYNTH** program (shown in Figure 22-7) demonstrates this idea:

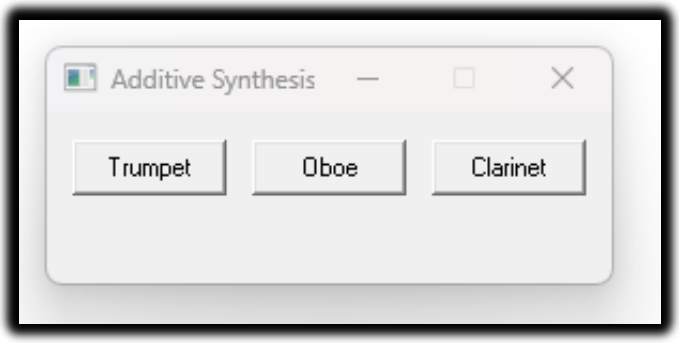
- It reads frequency and amplitude envelopes
- Generates multiple sine waves
- Adds them together
- Sends the result to the sound hardware

Using this method, sounds recorded decades ago can be recreated digitally.

6. Summary

- Timbre = distribution of partials + dynamic evolution.
- Fourier = decomposes complex tones into additive sine waves.
- Analog synthesis → limited, prone to drift → subtractive synthesis.
- Digital additive synthesis → accurate, controllable, captures inharmonicity.
- Attack phase & inharmonic partials = essential for realistic sound.

ADDSYNTH.C PROGRAM

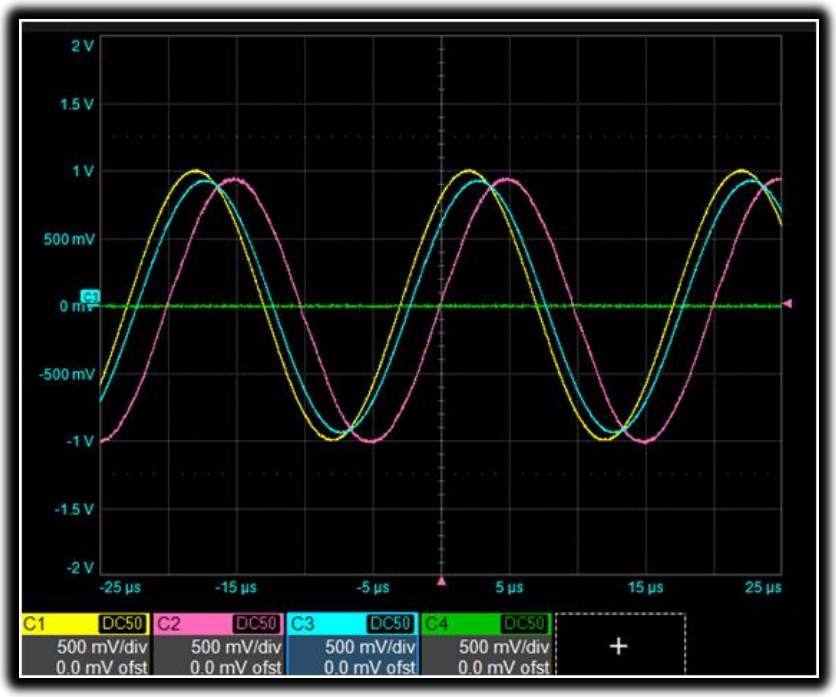


Sine Wave Generation

The SineGenerator function does exactly what its name suggests: it generates a sine wave for a given frequency.

It keeps track of a **phase angle**, which simply remembers where the wave left off between samples. This is what prevents clicks and breaks in the sound. Each time the function runs, the phase advances, the sine value is calculated, and a smooth waveform continues.

This function exists to generate **one harmonic at a time**. Everything else in the program is built on top of this simple idea.



Filling the Audio Buffer

The FillBuffer function is where additive synthesis actually happens.

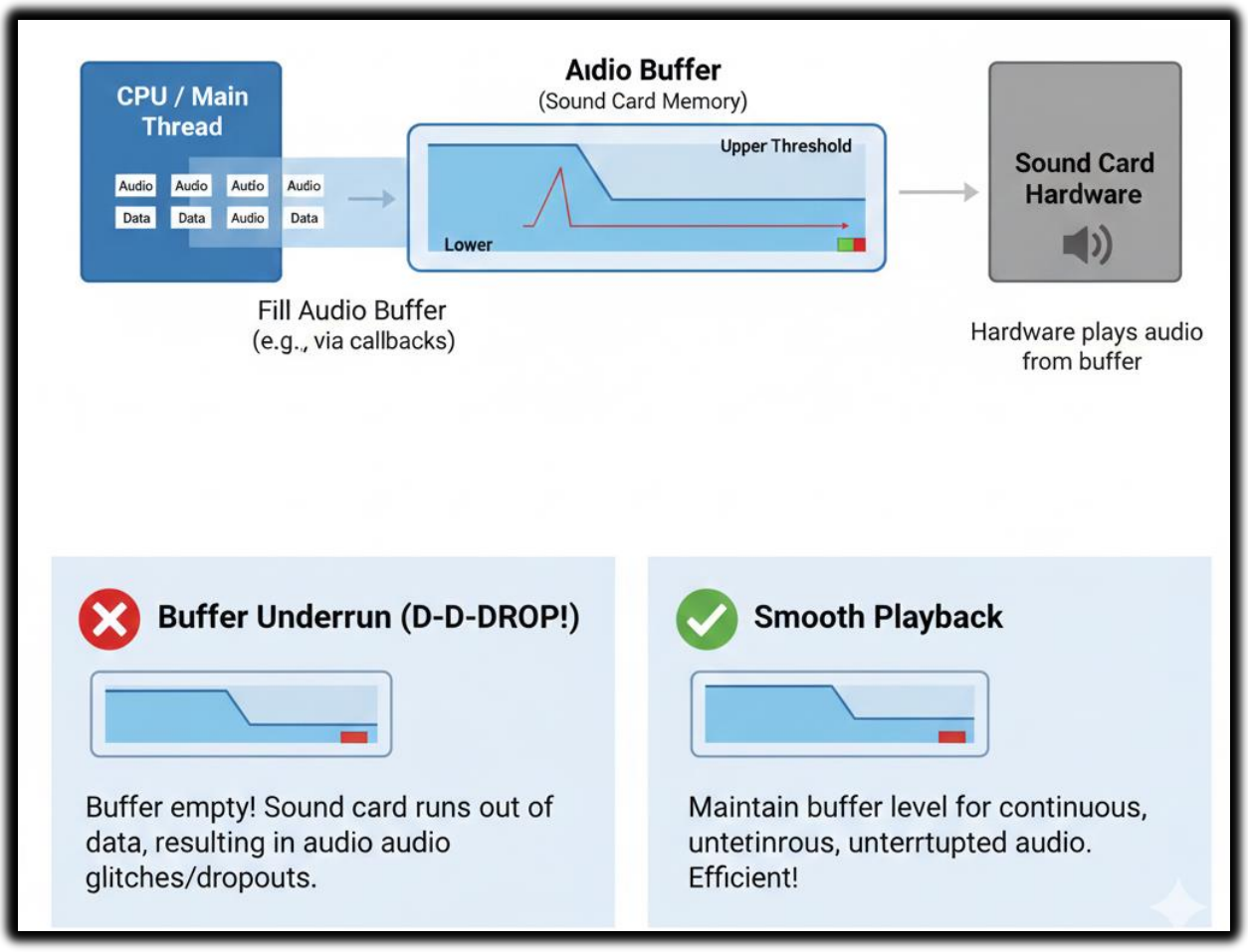
Instead of producing just one sine wave, this function:

- Walks through all the partials
- Figures out the current amplitude and frequency for each one
- Generates the sample value for each partial
- Adds them together

Amplitude and frequency envelopes control how each partial changes over time. The envelopes are just curves that say *"how strong"* and *"how high"* each component should be at a given moment.

All of these contributions are summed into a **single composite sample**, which is stored in an audio buffer. That buffer is later sent to the sound system or written to a file.

This is classic additive synthesis: build complex sound by stacking simple sine waves.



Writing the WAV File

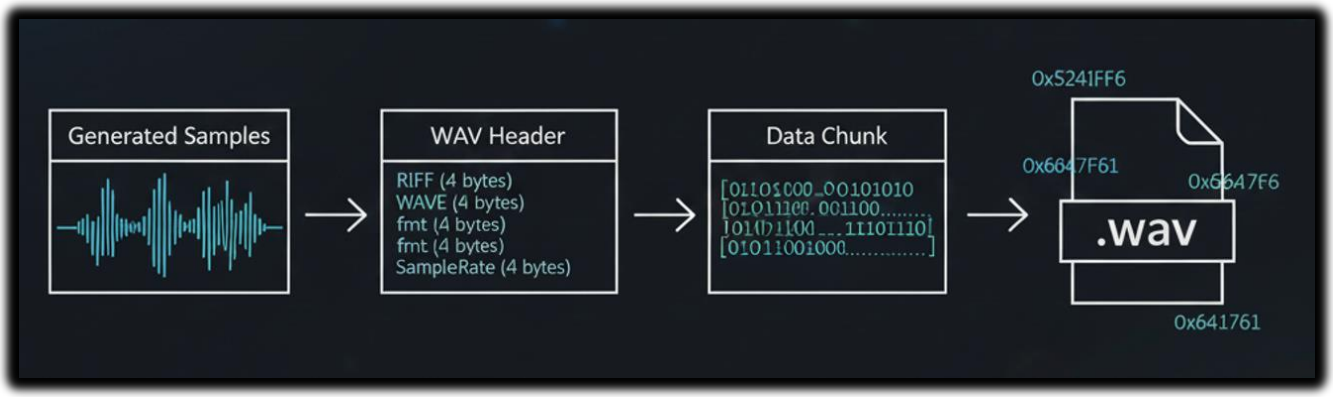
The MakeWaveFile function handles turning generated samples into a real .WAV file.

It:

- Creates the file using CreateFile
- Writes the required WAV headers
- Writes the sample buffer into the data chunk

Nothing fancy is happening here. The function just follows the WAV format rules so the file can be played by standard audio software.

By the time this function runs, all sound generation is already done.



User Interface Overview

The program uses a simple **dialog-based Windows UI**.

The main elements are:

- Buttons for trumpet, oboe, and clarinet
- A text area that shows preparation status

Each instrument button plays a pre-generated sound. Buttons are enabled only after the sound files are ready, so the user can't play something that doesn't exist yet.

The UI is minimal and functional—it's just a control panel for the synthesis engine.

Instrument Definitions

Each instrument (trumpet, oboe, clarinet) is defined by its own structure:

- insTrum
- insOboe
- insClar

These structures contain:

- Amplitude envelopes
- Frequency envelopes
- Partial definitions

Together, these values describe how the instrument's sound evolves over time. The envelopes are what make the sound feel alive instead of static.

Testing and Playback

Once a waveform file is successfully created:

- The corresponding button becomes active
- Pressing the button plays the sound

Playback is handled with PlaySound, using synchronous playback so the sound plays immediately when clicked.

Timers and Cursor Feedback

The program uses a timer (ID_TIMER) to manage background work without freezing the UI.

While heavy calculations are running:

- The cursor may change to a wait cursor
- The UI stays responsive

This gives the user visual feedback that work is in progress.

Resource Files

- ADDSYNTH.RC defines the dialog layout, buttons, and labels
- RESOURCE.H provides named constants for control IDs

This keeps UI code readable and avoids magic numbers.

Core Structures in ADDSYNTH.H

Three main structures define the synthesis model:

ENV - Stores time–value pairs for envelopes. Straight lines connect these points to form envelope curves.

PRT - Defines a partial. It stores:

- Number of envelope points
- Pointer to the ENV data

INS - Defines an instrument. It contains:

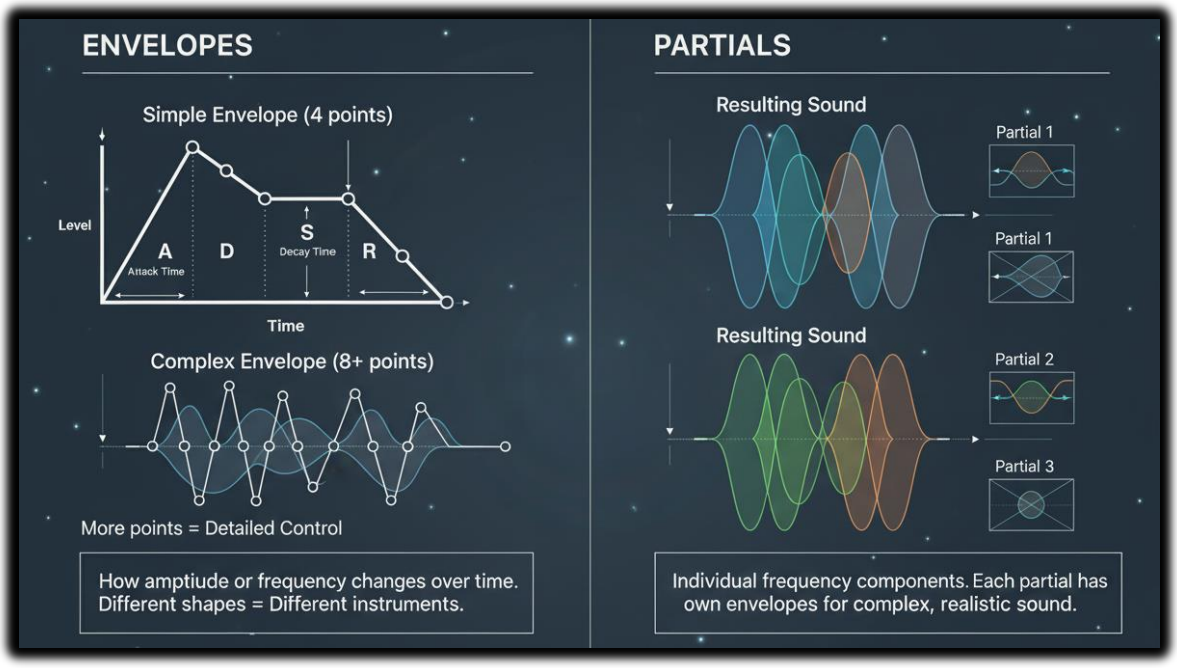
- Total duration
- Number of partials
- Pointer to the PRT array

These structures work together to describe how an instrument sounds over time.

Envelopes and Partials

Envelopes describe how amplitude or frequency changes during a note. Different instruments use different envelope shapes and numbers of points. More points mean more detailed control.

Partials are simply individual components of the sound. Each partial has its own envelope, allowing complex and realistic behavior.



Scaling and Maximum Amplitude

Before storing samples as 8-bit audio, the program calculates the **maximum possible combined amplitude** across all partials.

This value is used to scale the samples so:

- The sound is loud enough
- Clipping is avoided

This step ensures proper use of the available dynamic range.

Real-Time Limits and Button Enablement

Additive synthesis is computationally expensive.

Instead of trying to calculate everything in real time, the program:

- Precomputes the sound
- Enables buttons only when the work is done

This avoids UI freezes and makes the program usable on slower machines.

Reusing Existing Wave Files

On startup, the program checks whether waveform files already exist.

If they do:

- It skips regeneration
- Enables buttons immediately

This makes repeat runs faster and avoids unnecessary work.

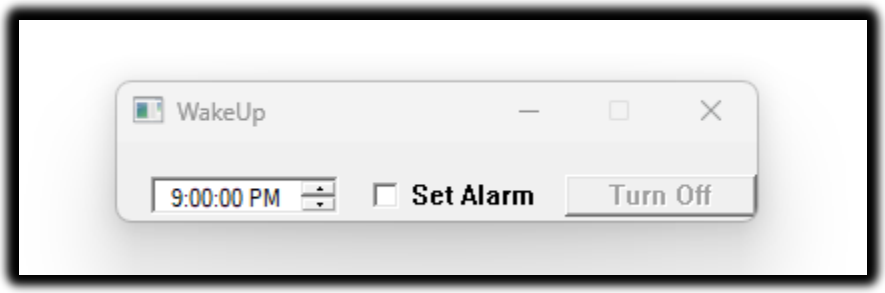
Final Note

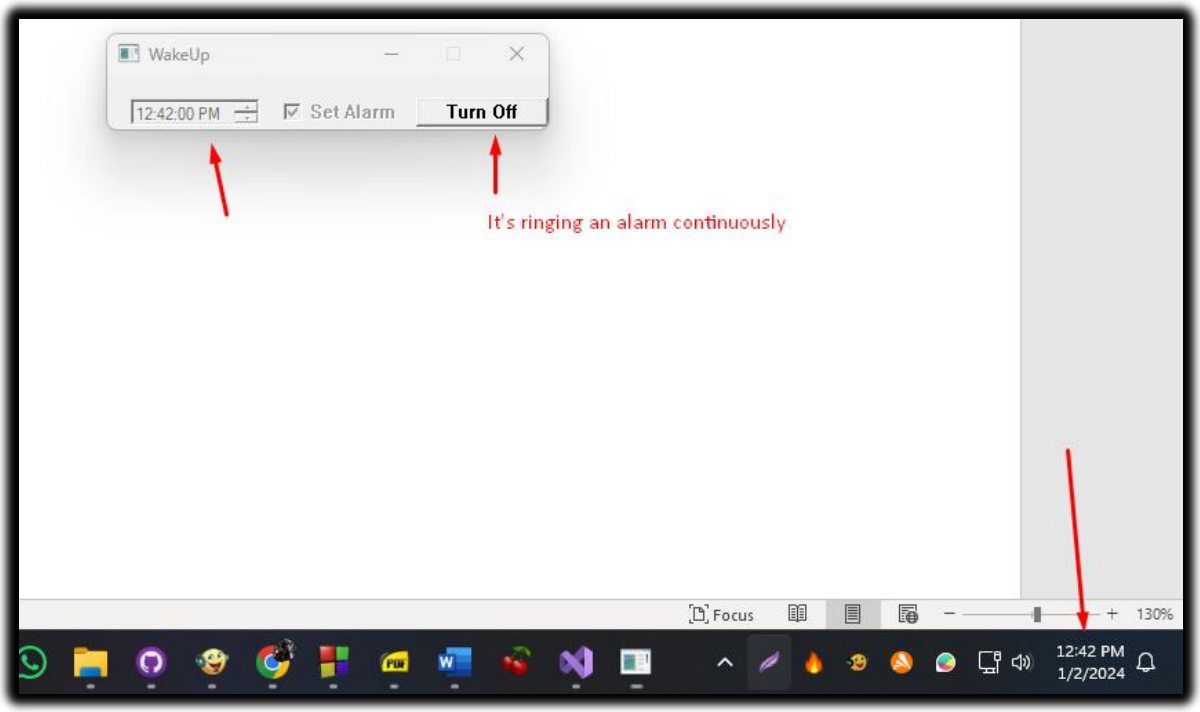
This program is less about flashy UI and more about demonstrating:

- Additive synthesis
- Envelope-based sound shaping
- Practical Windows audio programming

The code *is* the explanation—and now the notes finally match that reality 😌

WAKEUP ALARM CLOCK PROGRAM





Includes and Definitions

The program starts by including essential headers like windows.h and commctrl.h. These provide core Windows functions and common UI controls.

It also defines:

- Unique IDs for child windows (buttons, checkbox, date/time picker)
- Timer IDs
- Constants for waveform generation

These IDs and constants are like **name tags** for every component, so the program knows which control or timer you're talking about at any time.

Waveform Structure

The WAVEFORM structure represents a simulated audio file in memory. Its fields map closely to how a real .WAV file is organized:

WAVEFILEHEADER STRUCTURE ANALYSIS	
Field / Member	**Functional Purpose in RIFF/WAVE Format**
`chRiff[4]`	Identifies file as `"RIFF"`. The master wrapper for the multimedia resource.
`dwRiffSize`	Size of the entire RIFF chunk (Total File Size minus 8 bytes of header).
`chWave[4]`	The sub-format identifier. Must be `"WAVE"` to signify audio data.
`chFmt[4]`	Start of the Format Chunk: `"fmt "`. (Note the required trailing space).
`dwFmtSize`	Size of the format data block (usually 16 bytes for PCM).
`pwf`	**PCMWAVEFORMAT** structure: defines SamplesPerSec, Channels, and BitsPerSample.
`chData[4]`	Start of the Raw Data Chunk: `"data"`.
`dwDataSize`	The size of the following binary waveform sample data in bytes.
`byData[0]`	Flexible array (pointer) to the actual PCM waveform samples in memory.

Think of this structure as a **map of the audio file**, with each field pointing to where important info or data is stored.

Window Procedure and Subclassing

WndProc is the **message hub** for the program. It handles messages like:

- WM_CREATE – window creation
- WM_COMMAND – user actions (button clicks, checkbox)
- WM_NOTIFY – notifications from child controls
- WM_DESTROY – cleanup before exit

Three child windows (date/time picker, checkbox, push button) are **subclassed**. This means the program swaps in its own procedure (SubProc) to handle messages for those controls.
Think of subclassing like giving a control a **custom brain** so it can behave differently from the default.

Initialization and Child Windows

When WM_CREATE is received:

1. Window class information is registered
2. Main window is created
3. Common controls are initialized (InitCommonControlsEx)
4. A waveform file with alternating square waves is generated
5. Child windows are created:
    - ✓ **Date/time picker (hwndDTP)** – initialized to current time + 9 hours
    - ✓ **Checkbox (hwndCheck)** – set alarm
    - ✓ **Push button (hwndPush)** – turn off alarm

The date/time picker defaults to 9 hours ahead, giving a **ready-to-use alarm time**.

User Interaction Handling

WM_COMMAND manages all control interactions:

- **Set Alarm checkbox checked**
    - ✓ Calculates the difference between selected time and current PC time
    - ✓ Sets a **one-shot timer**
    - ✓ Plays alarm when timer expires
- **Checkbox unchecked**
    - ✓ Cancels the timer
- **Turn Off button clicked**
    - ✓ Stops the alarm sound
    - ✓ Resets the checkbox
    - ✓ Re-enables date/time picker

This ensures the program is **user-friendly**, letting you start, cancel, or turn off the alarm smoothly.

Date and Time Picker Handling

WM_NOTIFY handles notifications from the date/time picker (hwndDTP):

If the alarm time is changed while the alarm is active:

- Checkbox is unchecked
- Any running timer is canceled

This keeps the **display and actual timer in sync**, avoiding weird behavior.

Timer Handling

WM_TIMER handles the one-shot timer:

1. Timer is killed (so alarm only happens once)
2. Alarm sound starts playing
3. UI updates:
    - ✓ Date/time picker re-enabled
    - ✓ Checkbox re-enabled
    - ✓ Push button gets focus

This ensures the **user can interact immediately** after the alarm triggers.

FILETIME Structure and Large Integer Operations

- FILETIME uses dwLowDateTime and dwHighDateTime to store 64-bit intervals since **January 1, 1601**.
- __int64 allows math on 64-bit numbers
- LARGE_INTEGER union lets you treat it as:
    - ✓ Two 32-bit numbers
    - ✓ One 64-bit number

This is like **flexible math scaffolding** for working with big times and intervals.

Cleanup and Window Destruction

WM_DESTROY makes sure the program exits cleanly:

- Frees memory for waveform data
- Stops alarm if still playing
- Kills any active timer
- Resets the checkbox

No leftover sounds or timers — the program **leaves the system tidy**.

DEEP DIVE INTO MIDI AND MUSIC: BEYOND THE BASICS

MIDI (Musical Instrument Digital Interface) is much more than just "Note On" and "Note Off." It's a **language for digital instruments**, capable of transmitting expressive performance data and connecting diverse musical devices.

Beyond Note On and Note Off

MIDI messages go far beyond starting and stopping notes. Some key types:

1. **Pitch Bend** – Fine-tunes the pitch of a note for expressive vibrato or glides.
2. **Aftertouch** – Tracks pressure applied to a key after it's struck, allowing dynamic effects like vibrato or filter modulation.
3. **Control Change** – Modifies parameters like volume, pan, filter cutoff, and modulation.
4. **Program Change** – Switches between different sounds or patches on a synthesizer.
5. **SysEx (System Exclusive)** – Sends manufacturer-specific commands for advanced operations, like updating firmware or controlling custom hardware.

Analogy: If MIDI notes are letters in a musical sentence, these extra messages are punctuation and emphasis marks — they give music **expression and nuance**.

The Rise of Sequencers and Computer Integration

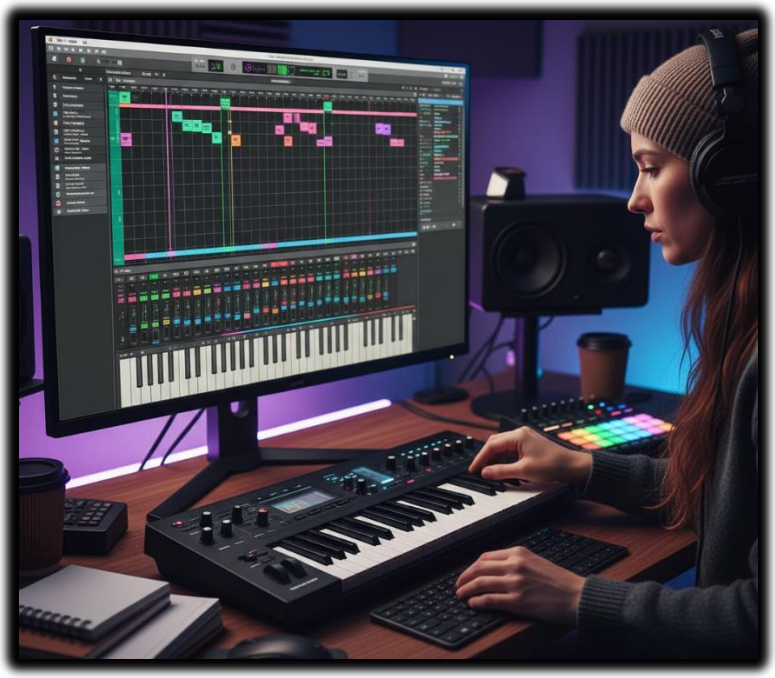
i) Standalone Sequencers

- Early tools for recording and editing musical sequences.
- Allowed composers to experiment with complex arrangements before computers were common.

ii) Computer-based Sequencers

- Software like **Pro Tools** or **FL Studio** revolutionized music production.
- Offered **flexible editing, layering, and mixing** capabilities, turning a computer into a full-fledged studio.

Impact: Sequencers made it possible to create professional music **without a full band**, democratizing music production.



Beyond Controllers and Synthesizers

MIDI is not just for keyboards or synths. It can control:

- Drums, guitars, wind instruments, even vocals
- Virtual instruments via computers
- Complex setups with multiple instruments

Key point: MIDI acts as a **universal translator** between instruments, controllers, and computers, expanding creative possibilities.

The Impact of MIDI

1. **Democratized Music Creation** – Lowered costs and simplified tools allowed amateurs and indie musicians to produce music.
2. **Birthed New Genres** – Techno, house, and other electronic genres wouldn't exist without MIDI.
3. **Revolutionized Live Performance** – One controller can manage multiple instruments and soundscapes.
4. **Preserved Musical Heritage** – MIDI files serve as **digital sheet music**, ensuring compositions can be reproduced or remixed.

MIDI is same to WinAPI Resource Scripts (Conceptually)

Just like a WinAPI resource file:

- Does not contain pixels
- Describes what to create (menus, dialogs, controls)
- Is interpreted by Windows to produce a UI

MIDI:

- Does not contain sound
- Describes what to play (notes, timing, velocity, instrument)
- Is interpreted by hardware or software to produce audio

Side-by-Side Analogy

RESOURCE COMPARISON: WINAPI VS. MIDI

FEATURE	WINAPI RESOURCE (.RC)	MIDI FILE (.MID)
Primary Content	Menu, Dialog, Icon, and String definitions.	Notes, tempo, velocity, and channel data.
Control Logic	Uses **IDs and Flags** (e.g., IDM_OPEN, WS_VISIBLE).	Uses **Program Changes** and **Control Changes** (CC).
Interpreter	Interpreted by the **Windows OS** kernel/user modules.	Interpreted by a **Synthesizer Engine** (Hardware or Software).
Final Output	Produces a **User Interface** (visual interaction).	Produces **Sound** (auditory sequence).
Storage Logic	Binary data appended to the .EXE via Linker.	Standard MIDI File (SMF) chunks: MThd and MTrk.

So yes — **MIDI is an instruction stream**, not data in the final form.

Important Nuance

MIDI does **not** specify:

- How a piano *sounds*
- What waveform to use
- What samples to load

It only says things like:

- “Channel 1: play note 60”
- “Velocity 90”
- “Use instrument 0 (piano)”
- “Release after 500 ms”

The interpreter decides how that sounds.

That’s why:

- The same MIDI file sounds different on different devices
- Old Sound Blaster cards, General MIDI synths, and modern VSTs all produce different results

Hardware vs Software Interpretation

I. Hardware (classic)

MIDI → Sound card synth → Analog audio

- Fixed instrument ROM
- Consistent but limited

II. Software (modern)

MIDI → Software synth / sampler → Digital samples → WAV

- Highly flexible
- Arbitrary sound quality

III. One-Sentence Summary

MIDI is a declarative instruction format for music, interpreted by a sound engine, just like a resource script is interpreted by Windows to create a UI.

Looking Ahead

- **Wireless MIDI** simplifies setups
- **AI integration** opens doors to interactive composition
- MIDI continues to **evolve**, remaining a cornerstone of music technology
- It allows you to explore a synthesizer's full palette of sounds via **Program Change messages**.

Understanding Program Changes

1. **Accessing Diverse Sounds**
    - Synthesizers contain banks of sounds (voices, patches, instruments).
2. **Sending the Message**
    - Format: C0 pp
    - pp ranges 0–127, selecting the program number.
3. **Common Controls**
    - MIDI keyboards often include numbered buttons to trigger Program Changes directly.

Analogy: Think of Program Change messages as **switching presets** on a digital instrument — like changing the brush in a digital painting app.

The Program Number Challenge

- **Lack of Standardization** – The same program number may produce completely different sounds on different devices.
- **Unexpected Surprises** – Sending a Program Change without knowing the mapping can create unpredictable results.
- **MIDI File Compatibility Issues** – A MIDI file may sound different depending on the synthesizer or software playback.

Enter General MIDI (GM)

1. **Standardization Effort** – GM assigns specific sounds to specific program numbers across compliant devices.
2. **Widespread Adoption** – Both hardware and software support GM, ensuring consistency.
3. **Ensuring Compatibility** – Using GM-compliant patches or mapping your device guarantees MIDI files sound as intended.

Key Takeaways

- Program Change messages are essential for **exploring a synthesizer's sonic palette**.
- Understanding program number variations helps **avoid surprises**.
- General MIDI provides a **predictable framework** for sound selection.
- For MIDI files, GM compatibility ensures **consistent playback** across devices.

UNVEILING THE ORCHESTRA WITHIN: A DEEP DIVE INTO MIDI CHANNELS

MIDI channels are one of the most important ideas in MIDI. They let **many instruments share one cable** without getting mixed up.

Think of a MIDI cable like a highway, and MIDI channels like **lanes**. All the data travels together, but each instrument listens only to its own lane.

What MIDI Channels Are

1. One Cable, Many Conversations

- A single MIDI cable can carry data for **up to 16 instruments at the same time**.
- Each instrument listens on a **specific channel** (numbered 0–15).
- Messages sent on Channel 3, for example, are ignored by devices listening on Channel 1 or 2.

This allows multiple instruments to play together without interfering with each other.

2 Assigning Channels (Avoiding Chaos)

- Every MIDI device or sound is assigned a channel.
- A device only responds to messages sent on its assigned channel.
- This prevents situations where one keyboard note accidentally triggers every instrument.

Analogy:
It's like giving each musician in an orchestra their own sheet music. Everyone plays together, but only their own part.

The Status Byte: Where the Channel Lives

1. What the Status Byte Does

- Every MIDI message starts with a **status byte**.
- This byte tells two things:
    1. **What kind of message it is** (Note On, Program Change, etc.)
    2. **Which channel it belongs to**

The **lower 4 bits** of the status byte store the channel number (0–15).

2. Why This Matters

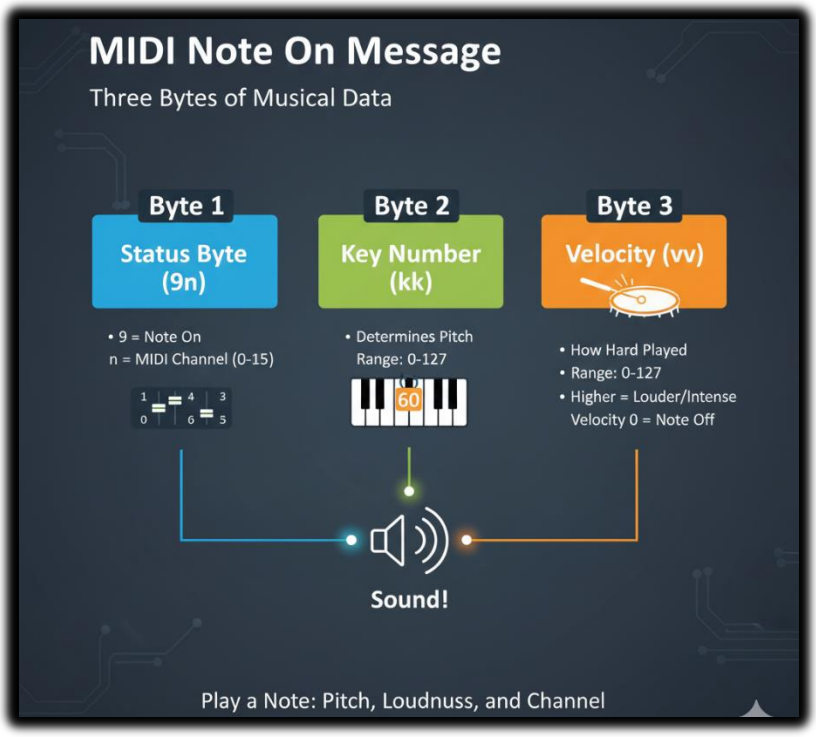
- The status byte acts like a **routing label**.
- It ensures the message reaches the correct instrument.
- Without it, MIDI devices wouldn't know who the message is for.

Deconstructing Common MIDI Messages

Note On Message

A Note On message starts a musical note and has **three bytes**:

1. **Status Byte (9n)**
    - 9 = Note On
    - n = MIDI channel (0–15)
2. **Key Number (kk)**
    - Determines the pitch
    - Range: 0–127
    - Middle C is usually **60**
3. **Velocity (vv)**
    - How hard the note is played
    - Range: 0–127
    - Higher value = louder or more intense sound
    - Velocity 0 is often treated as **Note Off**



Program Change Message

Program Change messages switch sounds or instruments.

They have **two bytes**:

1. **Status Byte (Cn)**
    - C = Program Change
    - n = MIDI channel
2. **Program Number (pp)**
    - Range: 0–127
    - Selects a sound from the device's sound bank
    - Example: **0 = piano**, **32 = guitar** (depends on the synth unless GM is used)

Channel Multiplexing (Why MIDI Is Powerful)

1. Simultaneous Control

Each MIDI channel works independently. Every channel can send:

- Note On / Note Off
- Control Change
- Program Change
- Pitch Bend

This allows **many instruments to play at once**.

2. Practical Example

- Channel 1 → Piano
- Channel 2 → Bass
- Channel 3 → Strings
- Channel 4 → Drums

All of this data travels through **one MIDI cable**, but each instrument hears only its own messages.

Typical MIDI “Conversation Flow”

Step 1: Program Changes First

- Program Change messages are usually sent at the start.
- They assign an instrument sound to each channel.
- This is like deciding who plays violin, trumpet, or piano.

Step 2: Notes Begin

- Note On and Note Off messages follow.
- Notes play on their assigned channels.
- Multiple channels playing together create harmony and layering.

Step 3: Changing Sounds Mid-Performance

- Program Change messages can also be sent **during playback**.
- This allows instruments to switch sounds in real time.
- Common in live performances and dynamic compositions.

One Voice per Channel (The Rule of Clarity)

- Each MIDI channel typically carries **one instrument voice** at a time.
- This keeps arrangements clean and predictable.
- It avoids confusion when editing or performing MIDI music.

Why MIDI Channels Matter

MIDI channels make it possible to:

- Layer multiple instruments
- Build complex arrangements
- Control many sounds from one controller
- Keep performances organized and expressive

They are the **foundation of structured MIDI music**.

Final Takeaway

MIDI channels are **independent communication lanes** inside a MIDI system. Understanding how channels and MIDI messages work together allows you to:

- Control the right instrument
- Avoid unwanted sound conflicts
- Build rich, multi-instrument compositions

Mastering MIDI channels turns a simple cable into a **full orchestra**—controlled, expressive, and powerful.

UNLEASHING THE MULTI-INSTRUMENTAL POWER OF MIDI CHANNELS

MIDI channels are the secret to making a single synthesizer sound like a full band. By utilizing the 16 available channels, we can split, layer, and conduct complex arrangements.

1. Transforming a Single Keyboard

You can use channels to alter how a keyboard behaves in real-time.

- **The Duet (Layering):**
  - ✓ You send two Program Change messages: C0 01 (Channel 1 gets Instrument 2) and C1 05 (Channel 2 gets Instrument 6).
  - ✓ **Action:** When you press one key, the keyboard sends **two** "Note On" messages—one for Channel 1 and one for Channel 2.
  - ✓ **Result:** You hear two instruments playing in unison (e.g., Piano + Strings).
- **The Split (Independent Control):**
  - ✓ You divide the keyboard into zones.
  - ✓ **Lower Keys:** Mapped to Channel 1 (e.g., Bass).
  - ✓ **Upper Keys:** Mapped to Channel 2 (e.g., Piano).
  - ✓ **Result:** Your left hand plays bass lines while your right hand plays melody. The keyboard acts as two separate instruments.

2. Orchestrating a 16-Piece Band (The Sequencer)

Using a PC running **Sequencing Software**, you become a virtual conductor.

- **The Concept:** The software assigns specific tracks to specific channels (Track 1 -> Flute on Ch 1; Track 2 -> Violin on Ch 2).
- **The Efficiency:** All this data travels down a **single MIDI cable**. The cable carries a stream of interleaved messages. The synthesizer at the other end reads the channel number on each message and routes it to the correct internal sound engine.

DECODING THE MIDI ORCHESTRA: A DEEP DIVE

Below is the reference table for the standard Channel Voice Messages we are discussing.

MIDI Messages:

MIDI Message	Status Byte	Data Bytes	Values
Note Off	8n	kk vv	kk = key number (0-127), vv = velocity (0-127)
Note On	9n	kk vv	kk = key number (0-127), vv = velocity (1-127, 0 = note off)
Polyphonic After Touch	An	kk tt	kk = key number (0-127), tt = after touch (0-127)
Control Change	Bn	cc xx	cc = controller (0-121), xx = value (0-127)
Channel Mode Local Control	Bn 7A	xx	xx = 0 (off), 127 (on)
All Notes Off	Bn 7B	00	
Omni Mode Off	Bn 7C	00	
Omni Mode On	Bn 7D	00	
Mono Mode On	Bn 7E	cc	cc = number of channels
Poly Mode On	Bn 7F	00	
Program Change	Cn	pp	pp = program (0-127)
Channel After Touch	Dn	tt	tt = after touch (0-127)
Pitch Wheel Change	En	ll hh	ll = low 7 bits (0-127), hh = high 7 bits (0-127)

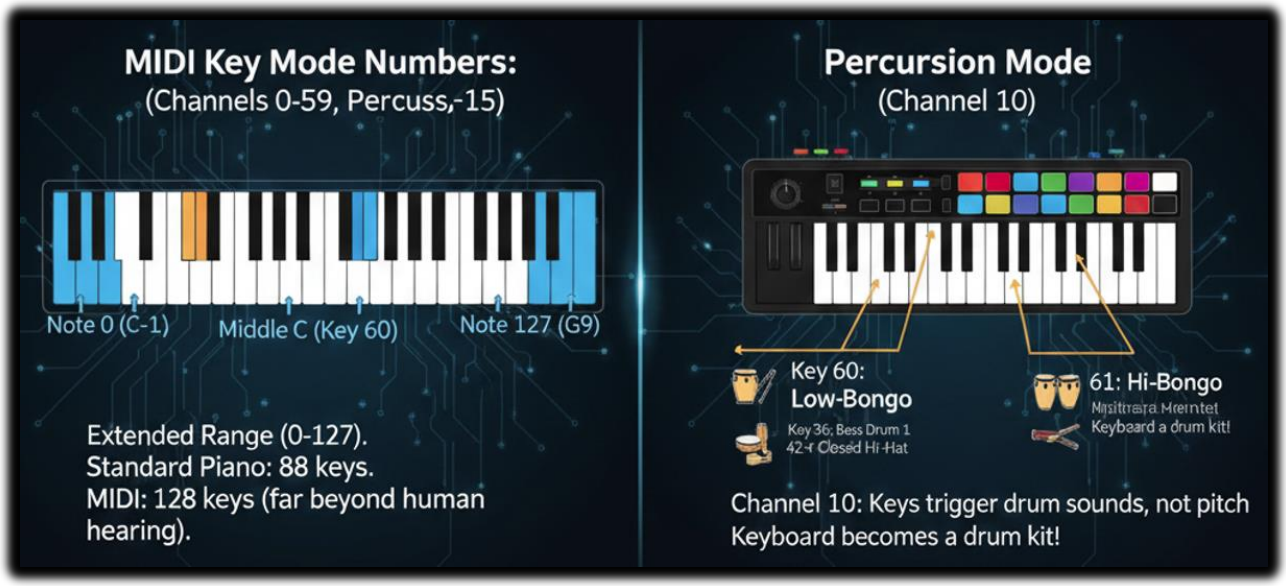
(Note: 'n' represents the Channel Number 0-F)

The Nuances of Expression

MIDI is about more than just which note is on or off. It captures the *soul* of the performance.

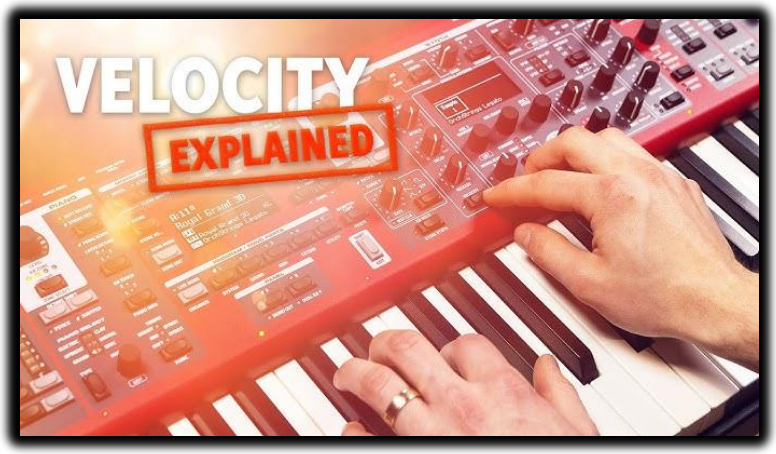
1. Key Numbers (Pitch & Percussion)

- **Extended Range:** A standard piano has 88 keys. MIDI supports **128 notes** (0–127). This allows for notes far lower and higher than a human can hear, often used for control signals or non-melodic data.
- **Percussion Mode:** On **Channel 10** (typically), key numbers don't represent pitch. Instead, Key 60 might be a "Hi-Bongo" and Key 61 a "Low-Bongo." The keyboard becomes a drum kit.



2. Velocity (The Sculptor)

- **More than Volume:** Velocity (0–127) measures how *fast* a key travels down.
- **Timbre Change:** On a real piano, hitting a key harder makes it brighter, not just louder. High-quality synthesizers use velocity to trigger different samples or open filters to replicate this aggression.



3. Aftertouch (The Squeeze)

Aftertouch is the pressure you apply *after* the key is already down.

- **Channel Aftertouch (Common):** If you press harder on *any* key, the effect (like vibrato) applies to *all* active notes on that channel.
- **Polyphonic Aftertouch (Rare):** Sensors detect pressure on *individual* keys. You can add vibrato to just the melody note while holding a chord steady.

4. Controllers (CC Messages)

These are the knobs, wheels, and sliders.

- **Standard CCs:** Volume (CC 7), Pan (CC 10), Sustain Pedal (CC 64).
- **Channel Mode Messages:** These are special CC messages (usually CC 120+) that tell the synth how to behave globally (e.g., "Stop all notes" or "Switch to Mono mode").

5. Pitch Bend

High Resolution: Unlike other controllers (which have 128 steps), Pitch Bend uses two data bytes to create **16,384 steps**. This ensures smooth, siren-like glides without hearing "stairstep" jumps in pitch.

System Messages: The Conductor's Baton

These messages (starting with **F0–FF**) do not belong to any specific channel. They control the entire MIDI system.

Real-Time Messages: These keep devices perfectly in sync.

- *Clock:* Keeps the tempo.
- *Start/Stop/Continue:* Controls the playback of a sequencer or drum machine.
- **Active Sensing:** A "heartbeat" message sent every few milliseconds. If a synth stops receiving this, it knows the cable was unplugged and shuts off all notes to prevent a "stuck note" drone.

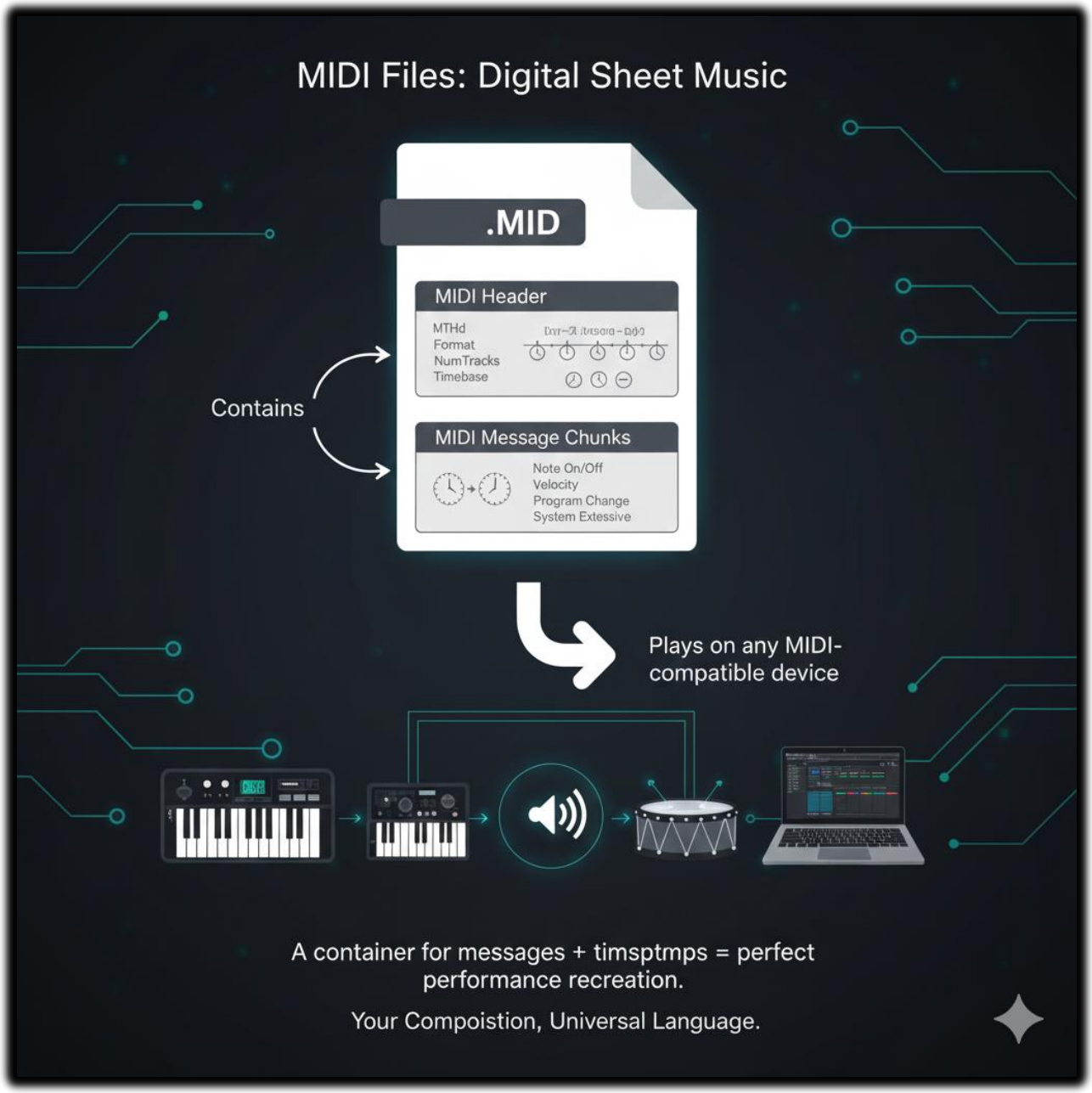
System Exclusive (SysEx): The "Secret Handshake."

- These messages allow manufacturers to send data that only *their* devices understand.
- **Use Case:** Backing up specific synthesizer patches to a computer or deep-editing sound parameters that standard MIDI CCs cannot reach.

MIDI Files

A MIDI file (.mid) is simply a container. It saves all these messages—Notes, Velocities, Program Changes, and System messages—along with **timestamps**.

It is the digital sheet music that allows a performance to be recreated perfectly on any MIDI-compatible device.



INTO MIDI SEQUENCING: ORCHESTRATING MUSIC WITH CODE

To play music programmatically, we bypass the high-level media players and talk directly to the MIDI hardware using the **Low-Level MIDI API**.

1. Opening the Connection: midiOutOpen

Before sending any music, you must open a line of communication with the hardware.

- **Function:** midiOutOpen(&hMidiOut, uDeviceID, ...)
- **The Handle:** If successful, this function gives you a handle (hMidiOut). You use this handle for all future commands.
- **Error Handling:** You must check the result. Common errors include:
  - ✓ MMSYSERR_ALLOCATED: The sound card is busy (another app is using it).
  - ✓ MMSYSERR_NODRIVER: No MIDI driver is installed.

2. Sending Messages: midiOutShortMsg

This function sends a single 32-bit package containing a MIDI command.

- **Packing the Data:** You must pack the Status Byte, Data Byte 1, and Data Byte 2 into a single DWORD variable using bitwise operations.
- **Example:** To play Middle C (Note 60) with max volume (Velocity 127) on Channel 1:
  - ✓ Status: 0x90 (Note On, Ch 1)
  - ✓ Data 1: 0x3C (Note 60)
  - ✓ Data 2: 0x7F (Velocity 127)
  - ✓ **Resulting DWORD:** 0x007F3C90
- **Closing:** When finished, always call midiOutClose to release the hardware.

THE BACHTOCC PROGRAM: A SEQUENCER CASE STUDY

This program plays the opening bars of Bach's *Toccata in D Minor* using code. It acts as a primitive sequencer.

1. The Data Structure (noteseq)

Instead of a MIDI file, the music is stored in a static C array called noteseq.

- **Format:** Each entry in the array represents one "event" or chord.
- **Fields:**
  - iDur: Duration (how long to wait before the next event).
  - iNote[0]: The first note to play.
  - iNote[1]: The second note to play (providing **Two-Note Polyphony**).

2. The Engine: WndProc & Timers

The program does not use a loop to play music (which would freeze the UI). Instead, it uses a **Windows Timer**.

- **WM_CREATE (Startup):**
  - Calls midiOutOpen to grab the **MIDIMAPPER** device.
  - Sends a **Program Change** message to switch the instrument to "Church Organ" (0xC0).
  - Calls SetTimer to start the heartbeat.
- **WM_TIMER (The Conductor):**
  - **Stop Previous Notes:** Sends "Note Off" commands for the notes that just finished playing.
  - **Check for End:** If the array is finished, call KillTimer.
  - **Play New Notes:** Reads the next row in noteseq and sends "Note On" commands for iNote[0] and iNote[1].
  - **Schedule Next Tick:** Resets the timer duration based on the iDur value of the *current* note. This allows fast notes and slow notes to mix.
- **WM_DESTROY (Cleanup):** Calls midiOutReset to silence any stuck notes. Calls midiOutClose.

3. The Limitation: Timer Resolution

The standard Windows SetTimer is based on the system clock tick. It is not accurate enough for professional music timing. Rhythms may feel "jittery" or uneven.

- **The Solution:** For serious music software, developers use **Multimedia Timers** (functions starting with timeSetEvent) which offer 1-millisecond precision.

4. Summary Checklist

1. **midiOutOpen**: Connects to the synth.
2. **midiOutShortMsg**: Sends the actual music notes.
3. **midiOutClose**: Disconnects.
4. **Sequencing Strategy**: Use a data array (Note/Duration) and a Timer to trigger notes sequentially without blocking the main program thread.

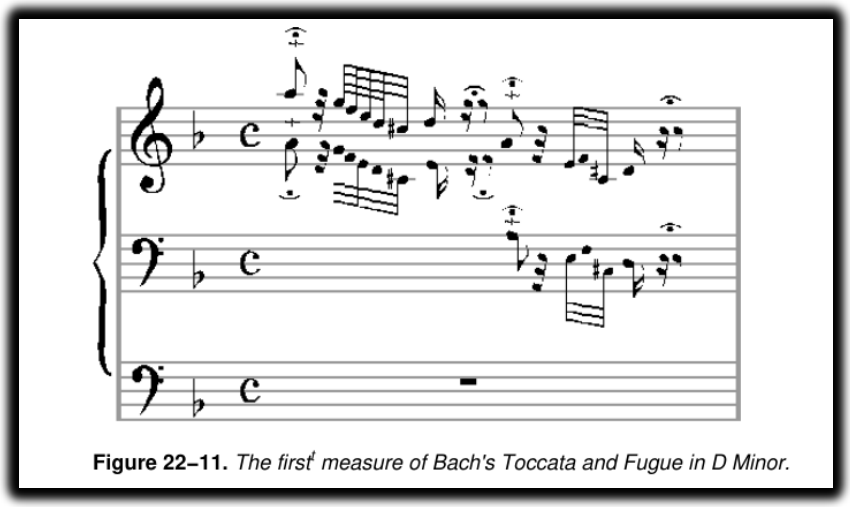


Figure 22–11. *The first measure of Bach's Toccata and Fugue in D Minor.*

BACCTOCC Program.mp4

KBMIDI PROGRAM IN-DEPTH

This section analyzes "KBMIDI," a program that turns your computer keyboard (QWERTY) into a musical instrument. It bridges the gap between Windows user inputs (keys, mouse, menus) and MIDI hardware.

1. The Communication Layer: The MIDI API

This is the engine room. The program uses three critical functions to talk to the sound card.

- **midiOutOpen (The Handshake):**
  - Connects the program to a specific MIDI output device (like your sound card's synth).
  - **Critical Step:** You must handle errors here. If the device is busy (locked by another app), the program must warn the user, not crash.
- **midiOutShortMsg (The Messenger):**
  - This is the workhorse. Every time you press a key or move a scrollbar, this function sends a 32-bit package.
  - **It handles:**
    - **Note On/Off:** Playing sounds.
    - **Program Change:** Switching from Piano to Guitar.
    - **Control Change:** Adjusting volume or pitch bend.
- **midiOutClose (The Cleanup):**
  - Called when the app exits. Failing to call this can "orphan" the MIDI device, locking it so no other apps (like Windows Media Player or a web browser) can use audio until you reboot.

2. The Brain: Window Procedure (WndProc)

The WndProc is the traffic controller. It listens for Windows messages and translates them into MIDI commands.

a) Handling the Keyboard (QWERTY -> Musical Notes)

The Mapping: The program has a lookup table. It knows that the 'Z' key might be Low C, and the 'S' key might be Low C#. The Logic goes this way:

WM_KEYDOWN: Look up the note -> Send Note On command -> **Paint Key Black** (Visual Feedback).

WM_KEYUP: Look up the note -> Send Note Off command -> **Paint Key White**.

Octaves: It watches for Modifier Keys (Shift or Ctrl). If Shift is held, it adds +12 to the note number to play an octave higher.

b) Handling Menus (Dynamic Creation)

- **The Problem:** You don't know what MIDI devices the user has installed.
- **The Solution:** At startup, the program asks Windows, "List all MIDI devices." It then builds the "Open" menu dynamically based on that list.
- **Functionality:** The menu allows switching **Instruments** (Voices). When you select "Violin," the program sends a Program Change message to the synthesizer.

c) Handling Scroll Bars (Expression)

Instead of just clicking notes, the user can "perform" using scroll bars.

- **Vertical Scroll (WM_VSCROLL):** Mapped to **Pitch Bend**. Moving it up/down bends the note sharp or flat.
- **Horizontal Scroll (WM_HSCROLL):** Mapped to **Velocity** (Volume) or Modulation.

3. The Face: Graphics and Visual Feedback

A musical instrument needs to feel responsive. KBMIDI uses graphics to mimic physical touch.

Drawing the Keys: The program calculates the size of the window and draws a piano keyboard proportional to the screen size.

Inversion (Feedback):

- When you press a key, the program creates a "Negative Image" of that specific key (White becomes Black, Black becomes White).
- This confirms to the user: "Yes, I registered your press."

4. Code Organization & Safety

Resource Management

- **Device Safety:** If a device fails to open, show a message box. Don't let the user play on a "null" device.
- **Memory:** If the program allocates memory for instrument lists, it frees it upon closing.

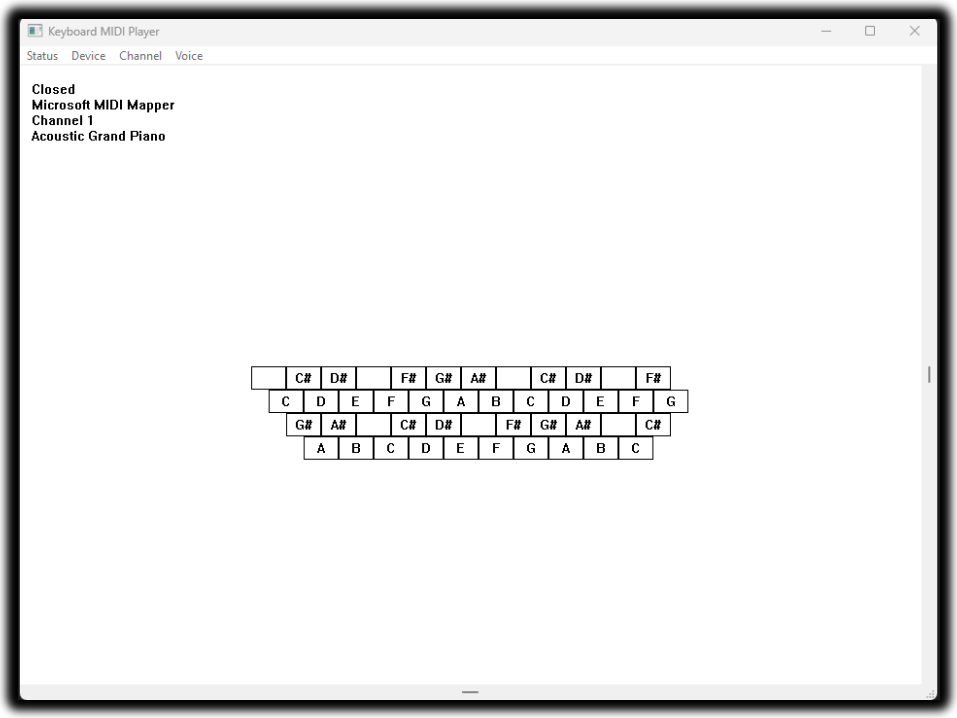
Structured Data

Instead of messy if/else statements, the program uses **C Structures (struct)** to organize data.

- **Example:** An Instrument *structure* might hold the Name ("Grand Piano") and the ID (0).
- **Benefit:** This makes it easy to add new instrument definitions without rewriting the core logic.

Summary Checklist

1. **midiOutOpen** connects; **midiOutClose** disconnects (never forget this!).
2. **WndProc** converts PC keyboard events (WM_KEYDOWN) into MIDI Sound events (Note On).
3. **Visual Feedback** (Inverting colors) is crucial for a good user experience.
4. **Dynamic Menus** prevent showing devices that don't exist.
5. **Scroll Bars** offer continuous control (Pitch/Velocity) rather than just on/off.



KBMIDI: TECHNICAL REFERENCE & USER MANUAL

We have built the engine; now let's look at the specific controls, mapping logic, and known quirks of the KBMIDI program.

1. The Keyboard Map (The Virtual Piano)

The program maps the standard QWERTY keyboard to musical notes starting at **A (110 Hz)**.

Base Position: The 'Z' key is mapped to Low A.

Range: The keys span 3 octaves (Bottom row up to the top row).

Octave Shifting:

- **Ctrl Key:** Shifts everything *down* 1 octave.
- **Shift Key:** Shifts everything *up* 1 octave.
- **Total Range:** 5 Octaves.

2. The "Hidden" Drum Mode

MIDI Channel 10 is reserved by the General MIDI standard for percussion.

How to access: Set the **Channel** menu to **10**.

Result: The "Voice" menu becomes irrelevant. Instead, every key on your keyboard plays a different drum sound (Snare, Kick, Hi-Hat, Cowbell) rather than a musical pitch.

3. The Scroll Bars (Expression Controls)

Since a PC keyboard is not pressure-sensitive, we use scroll bars to add expression.

a) Horizontal Scroll (Volume/Velocity):

- Adjusts how "hard" the notes are struck.
- *Technical:* MIDI Velocity (0–127).

b) Vertical Scroll (Pitch Bend):

- Acts like the "Whammy Bar" on a guitar.
- **Behavior:** Dragging it up increases pitch; dragging down lowers it. Releasing the mouse snaps it back to the center (neutral).
- **The Math:** Pitch Bend uses **14-bit precision** (two 7-bit bytes).
    - ❖ 0x0000: Full Bend Down.
    - ❖ 0x2000: Center (No Bend).
    - ❖ 0x3FFF: Full Bend Up.

4. Crucial "Quirks" & Bugs

These are important limitations of the Windows Message Loop in this specific program:

The "Scroll Bar Freeze":

- When you click and drag a scroll bar, Windows enters a "modal loop." This pauses the main program execution.
- **Effect:** If you are holding a key down (Note On) and then grab the scroll bar, the program stops listening to the keyboard. When you let go of the key, the program *won't hear* the release.
- **Result:** The note will hang (play forever).

The "Panic Button" (Esc):

- Because of the issue above, we added a fail-safe in the code.
- **Action:** Pressing **Esc** sends an "All Notes Off" message to all 16 channels, silencing any stuck notes immediately.

5. Final Code Summary

- **Dynamic Resources:** The menus are built at runtime using midiOutGetDevCaps, so the program never lists a device you don't actually have.
- **Patching:** When you change settings (Voice/Channel), the program calls a custom function MidiSetPatch to update the synthesizer immediately.

THE GRAND FINALE: THE "DRUM" PROGRAM

The **DRUM** application is a 47-channel step sequencer. Unlike the KBMIDI program which was for *performance* (playing in real-time), this is for *composition* (programming a loop).

1. The User Interface (The Grid)

The window is divided into a classic sequencer layout:

- **The Y-Axis (Rows):** Lists 47 percussion instruments (Bass Drum, Snare, Hi-Hat, etc.).
- **The X-Axis (Columns):** Represents time (Beats).
- **The Interaction:**
  - **Left-Click:** Adds a Percussion hit (Dark Gray).
  - **Right-Click:** Adds a Piano tone (Light Gray).
  - **Both:** Plays both (Black).
  - **Bouncing Ball:** A visual cursor moves across the grid to show the current beat.

---

2. Architecture: The triple threat

The code is split into three distinct modules to handle the complexity.

Module A: The GUI (DRUM.C)

This handles the window, painting, and user input.

WndProc: The central hub.

- **WM_LBUTTONDOWN:** detects which grid cell was clicked and updates the data structure.
- **WM_PAINT / DrawRectangle:** Instead of redrawing the whole window, it efficiently paints just the specific beat rectangles (Gray/Black/White) to keep the UI snappy.
- **WM_USER_NOTIFY:** A custom message sent *from* the audio engine to the GUI to say "Hey, I just played a beat, move the bouncing ball."

Module B: The Engine (DRUMTIME.C)

The Problem: The standard Windows SetTimer (used in the Bach example) is inaccurate. It drifts by milliseconds, which makes drums sound "sloppy" or off-beat.

The Solution: Multimedia Timers.

- **timeSetEvent:** This API requests a high-resolution timer interrupt (down to **1 millisecond** accuracy).
- **Threading:** This timer runs in its own thread, separate from the GUI. This ensures that even if you drag a window (which usually freezes WM_TIMER messages), the music keeps playing perfectly.
- **The Callback (DrumTimerFunc):** This function wakes up every beat, sends the MIDI commands, and tells the GUI to update.

Module C: The Storage (DRUMFILE.C)

We need to save our beats. We don't use a text file; we use the **RIFF (Resource Interchange File Format)**. This is the same structure used by .WAV and .AVI files.

Chunks: Data is stored in "Chunks." A chunk has a 4-letter tag (like "data" or "fmt ") and a size.

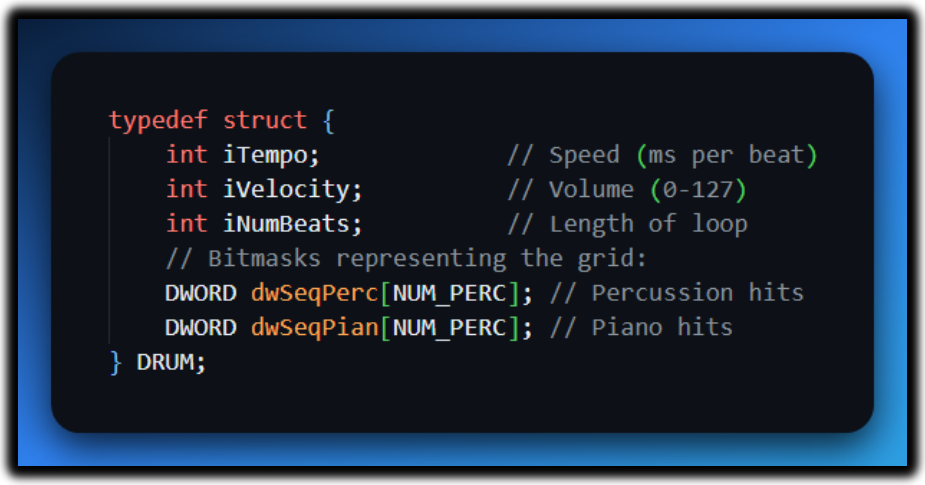
The Nesting:

- **RIFF Chunk:** The container.
- **LIST Chunk:** Contains Info (Title, Author).
- **data Chunk:** Contains the actual beat patterns.

mmio Functions: We use specific Multimedia I/O functions (mmioOpen, mmioDescend, mmioAscend) to navigate these chunks like a file system.

3. Critical data structures (drumtime.h)

The DRUM structure acts as the "Save File" for the application.



```
typedef struct {
    int iTempo;             // Speed (ms per beat)
    int iVelocity;          // Volume (0-127)
    int iNumBeats;          // Length of loop
    // Bitmasks representing the grid:
    DWORD dwSeqPerc[NUM_PERC]; // Percussion hits
    DWORD dwSeqPian[NUM_PERC]; // Piano hits
} DRUM;
```

HOW IT WORKS: THE LOGIC FLOW

Step 1: Initialization

- The app launches. DrumSetParams initializes the grid to empty.
- The user sees 47 instrument names (Bass Drum, etc.) drawn on the left.

Step 2: Composition

- The user clicks a square. WndProc calculates the coordinates to find which Instrument (Row) and Beat (Col) was clicked.
- It updates the dwSeqPerc bitmask (sets a 1 at that position).
- It calls DrawRectangle to turn that square gray.

Step 3: Playback (The Sequence)

1. User clicks "Running".
2. **DrumBeginSequence**: Opens the MIDI Mapper.
3. **timeBeginPeriod**: Tells Windows "I need high precision timing now."
4. **timeSetEvent**: Starts the heartbeat.
5. **DrumTimerFunc (The Heartbeat)**:
    - ❖ Checks the dwSeqPerc array for the current beat.
    - ❖ If a bit is set (1), it sends a midiOutShortMsg to play that drum.
    - ❖ Increments the beat counter.
    - ❖ Sends WM_USER_NOTIFY to the main window ("Update the bouncing ball!").

Step 4: Saving

- User clicks Save.
- DrumFileWrite opens a file.
- It writes the **RIFF** header.
- It dumps the DRUM structure into a **data** chunk.
- It closes the file using mmioClose.

USER MANUAL & CONTROLS

Tempo: Controlled by the **Vertical Scrollbar**.

- Top: Fast (10ms/beat).
- Bottom: Slow (1 sec/beat).

Volume: Controlled by the **Horizontal Scrollbar**.

Loop Length: The standard grid is 32 beats. You can shorten the loop by clicking above the grid. A repeat sign **(:|)** will appear, and the sequencer will loop back early.

Stop: Click "Stopped" in the menu. This calls timeKillEvent to stop the high-precision timer.

That's how windows handles sound at the silicon level 😎
We're done, chapter 23 is a 10-minute read.