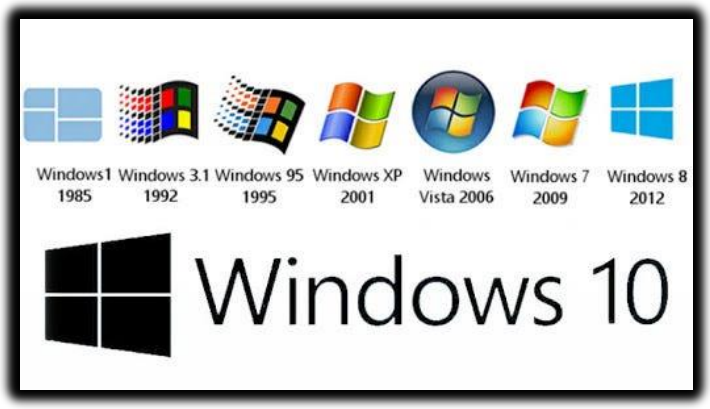
CHAPTER 22 – SOUND & MUSIC IN WINDOWS

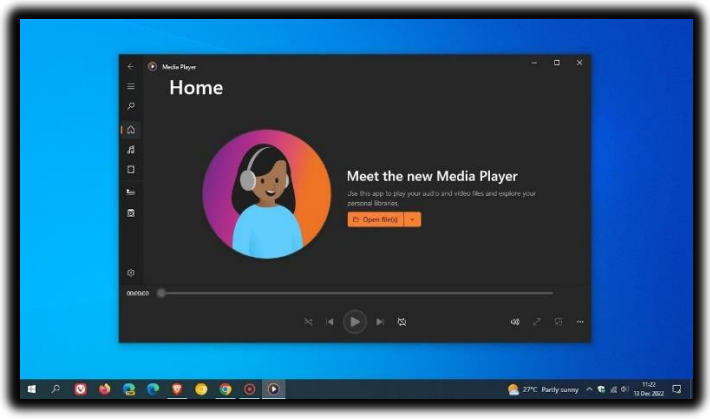
History & Context

- **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.

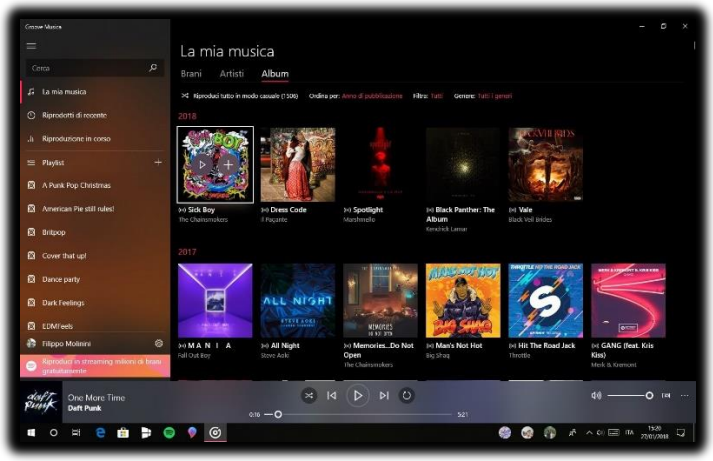
2. Built-in Windows 10 Tools



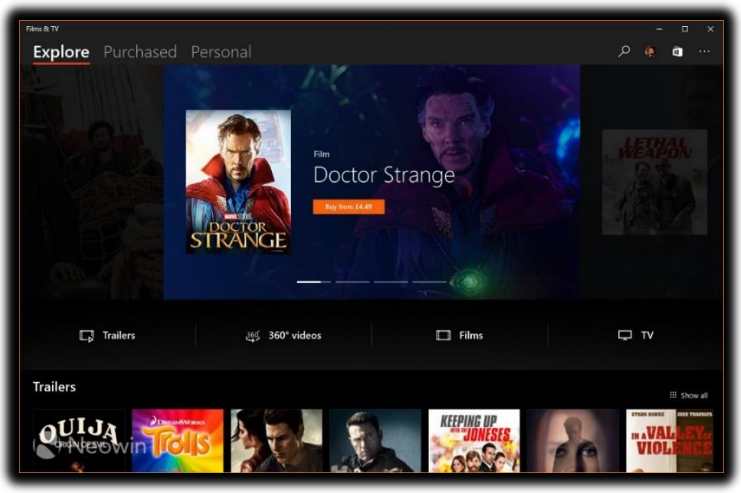
Windows Media Player: Play audio/video files, manage playlists, organize libraries.



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

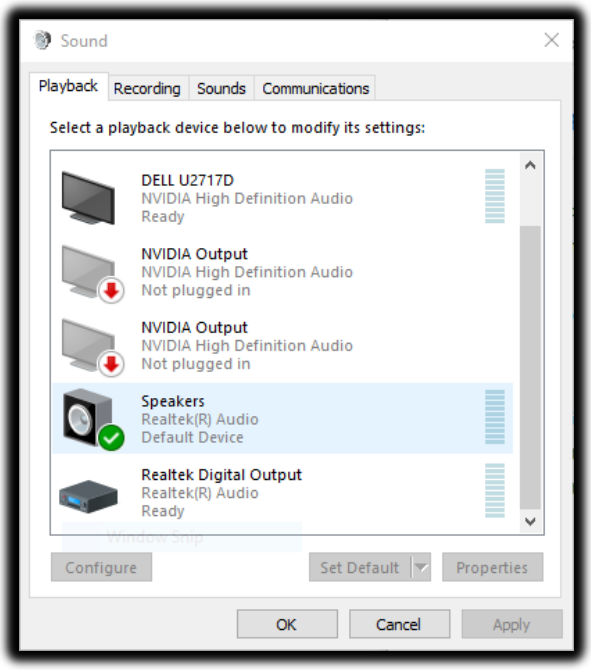


Movies & TV: Play videos, purchase/rent content, supports subtitles & casting.



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

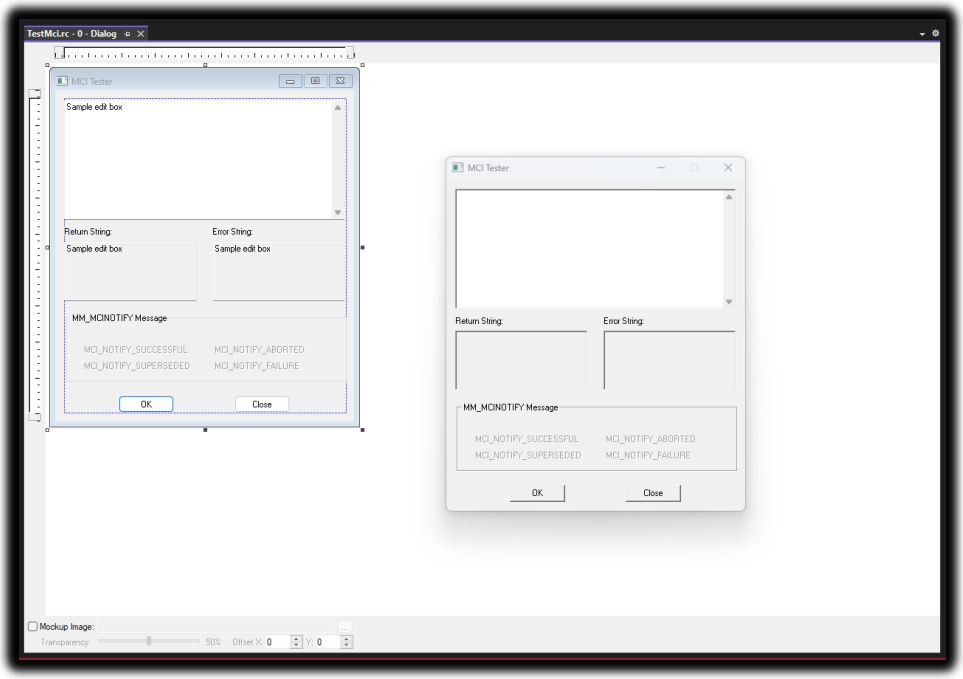
Windows gives **two tiers** of multimedia APIs:

- **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

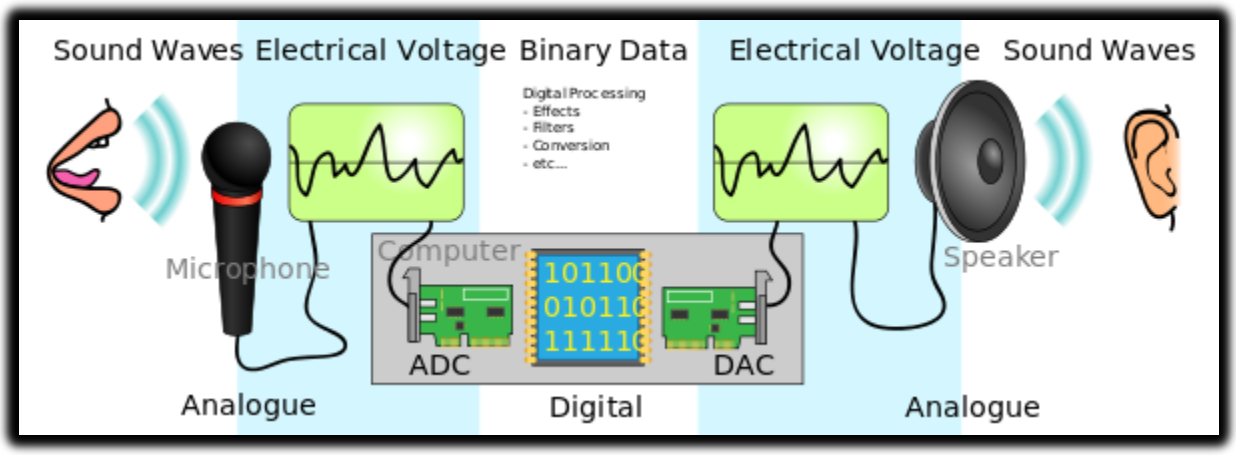
- Build a **mini-CD player UI**: display track info, play/pause, show a timer.
- **Sync graphics with music:** for instruction software or small visualizers.
- Automation of repeated tasks with **MCI scripts**.

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**.

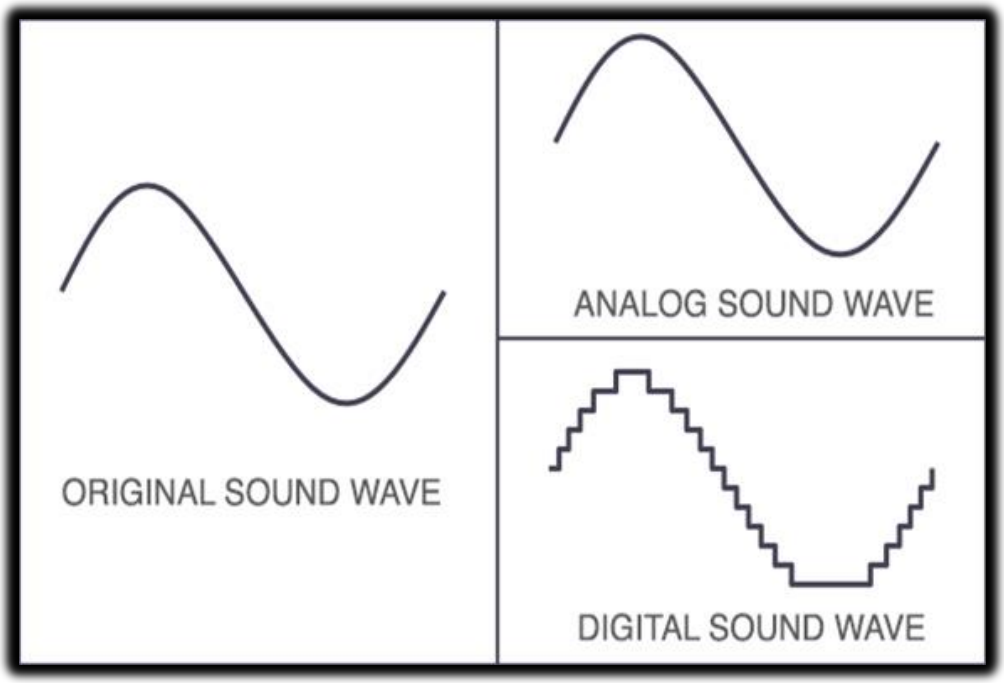
From sine waves → harmonics → timbre → digital storage, understanding this makes you **ready to leverage the Waveform API efficiently**.

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.

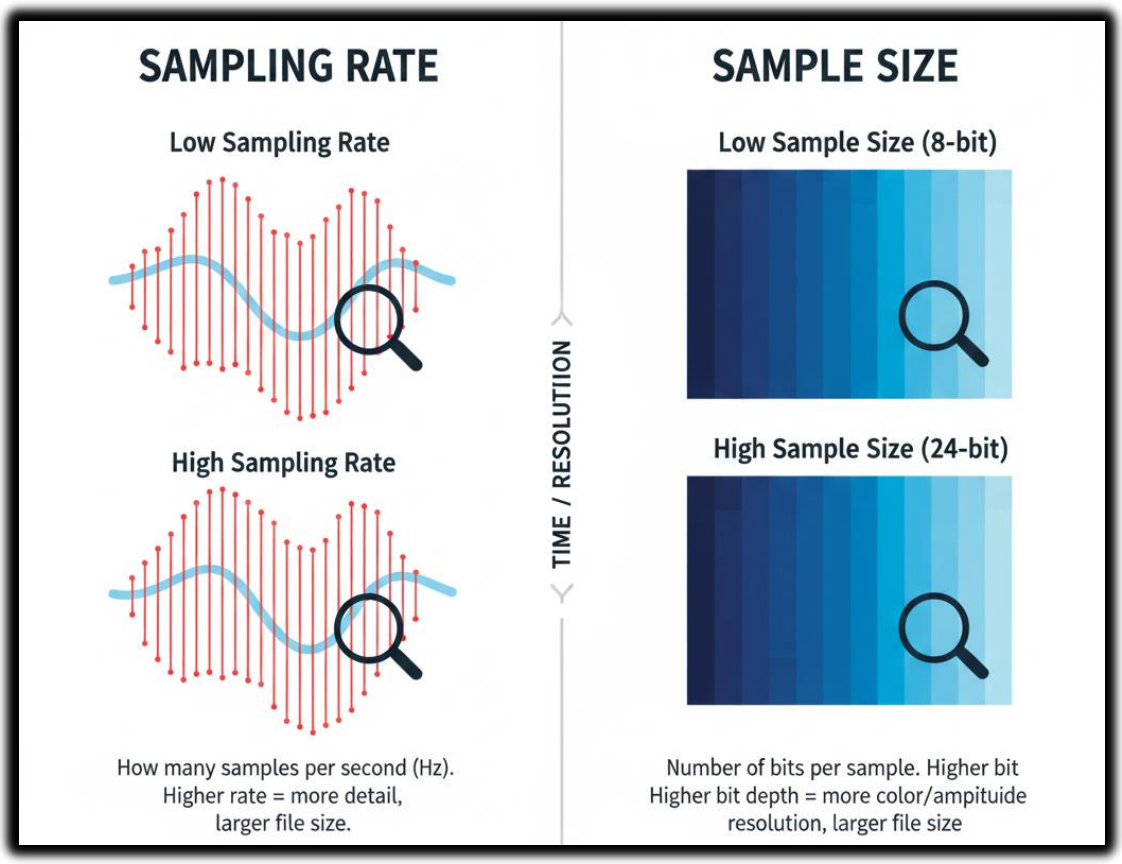


2. Sampling Rate & Sample Size

- **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**.

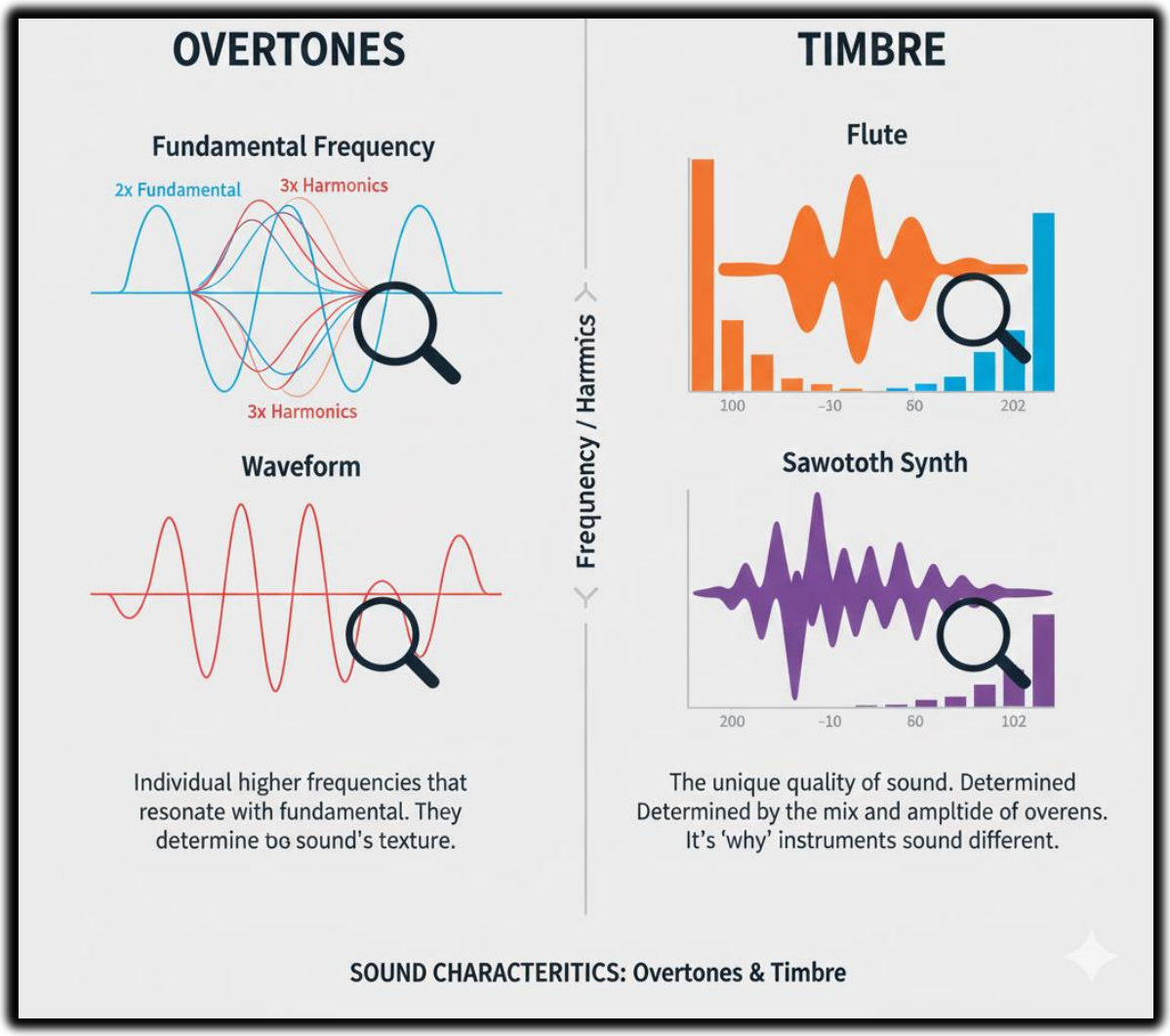


3. Overtones & Timbre

- 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 🎵

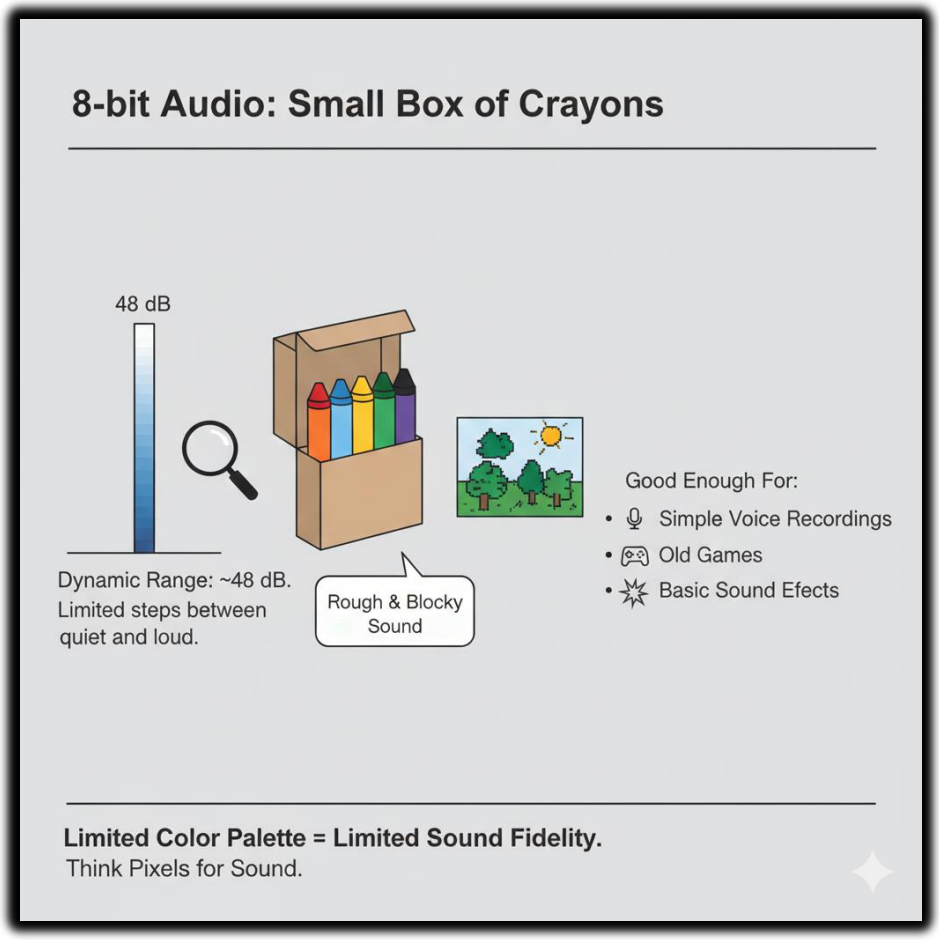
Common Sample Sizes and What They Mean

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

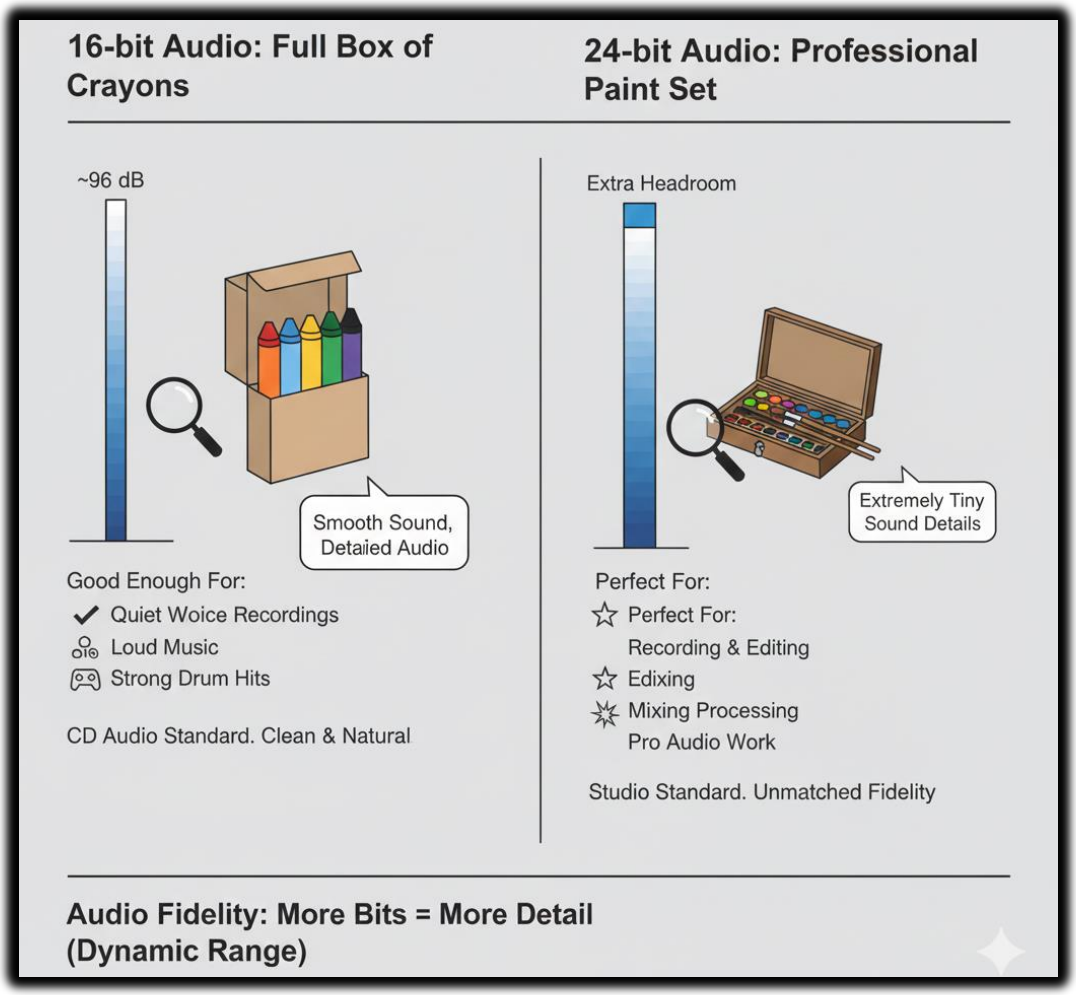
For most listening, 16-bit audio already sounds very clean and natural.

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.

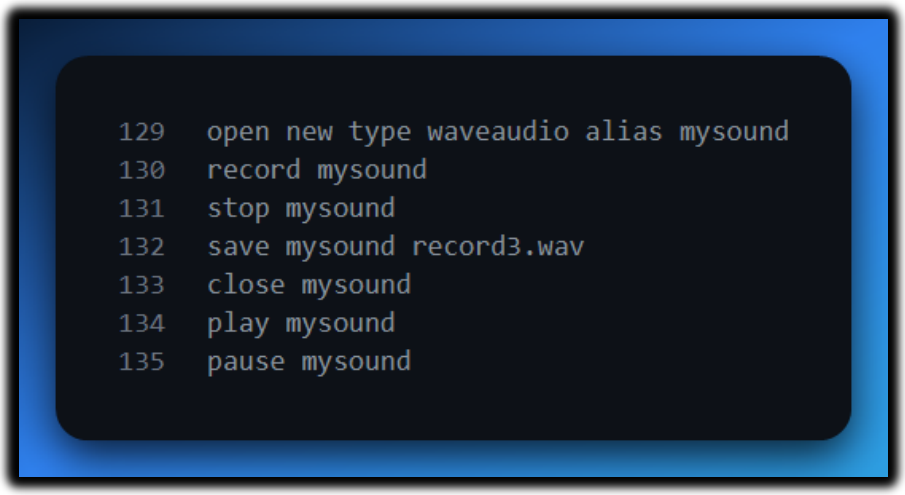
Cons:

- 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**.

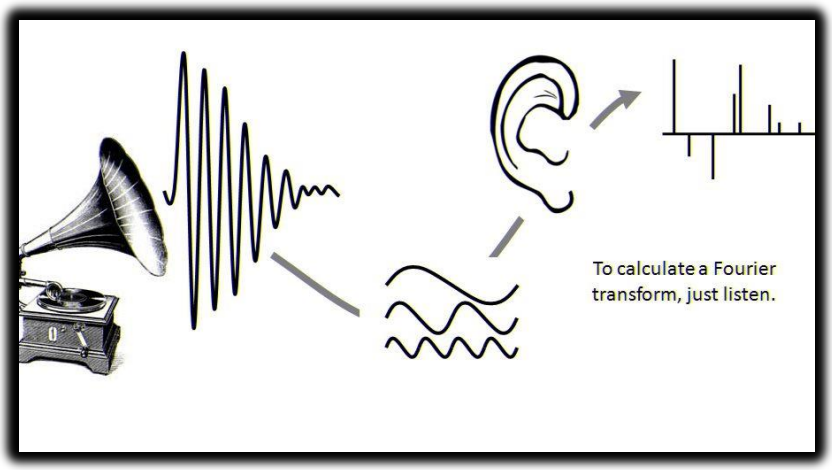
Timbre: Why Instruments Sound Different

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:

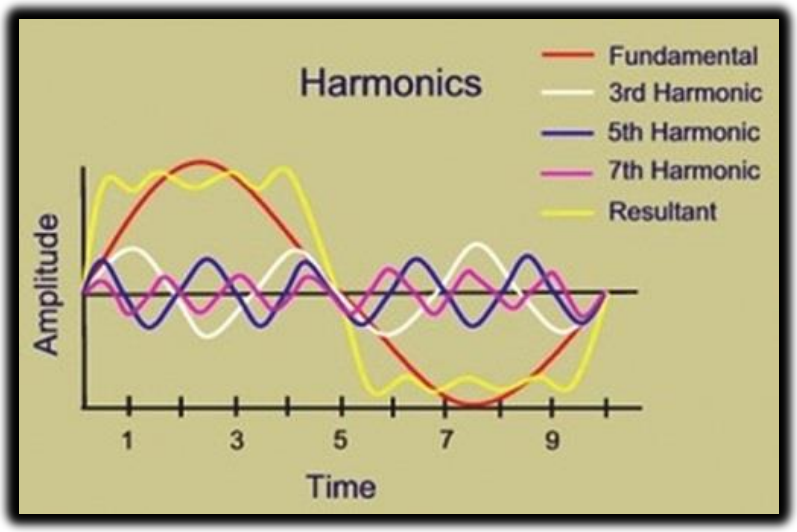
- Only odd harmonics
- Amplitudes that decrease like: 1, 1/3, 1/5, 1/7, ...

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.



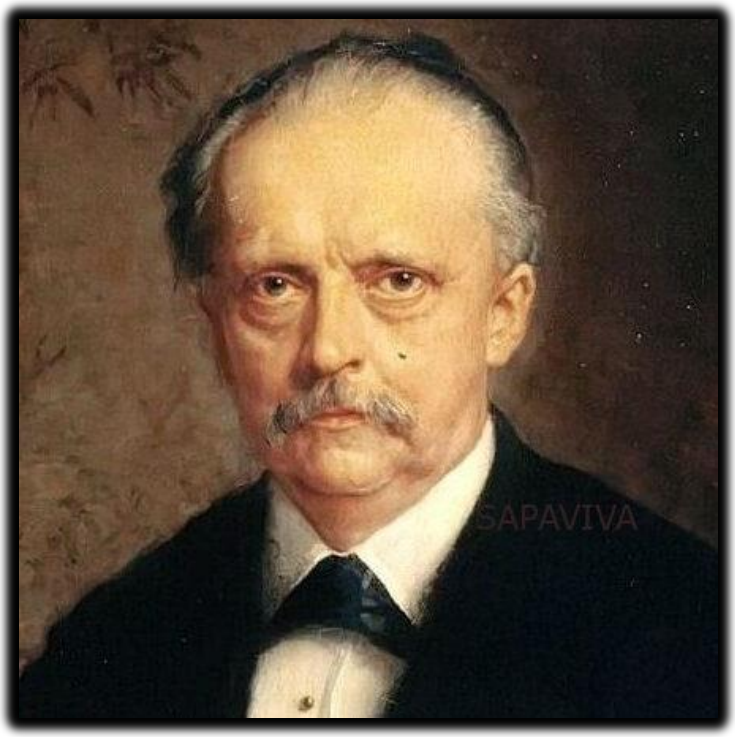
From Science to Music: Helmholtz and Early Theory

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.

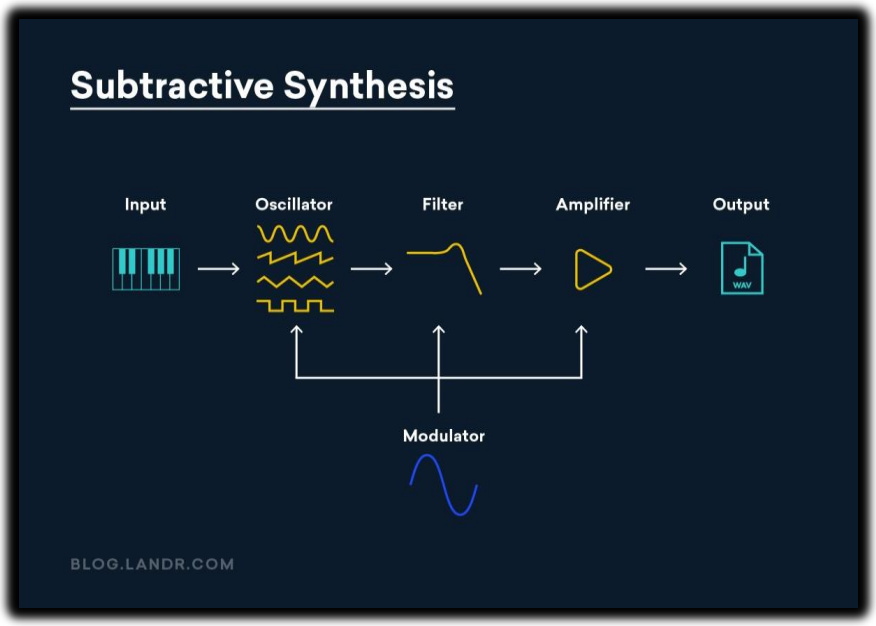
Filters and Subtractive Synthesis

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.

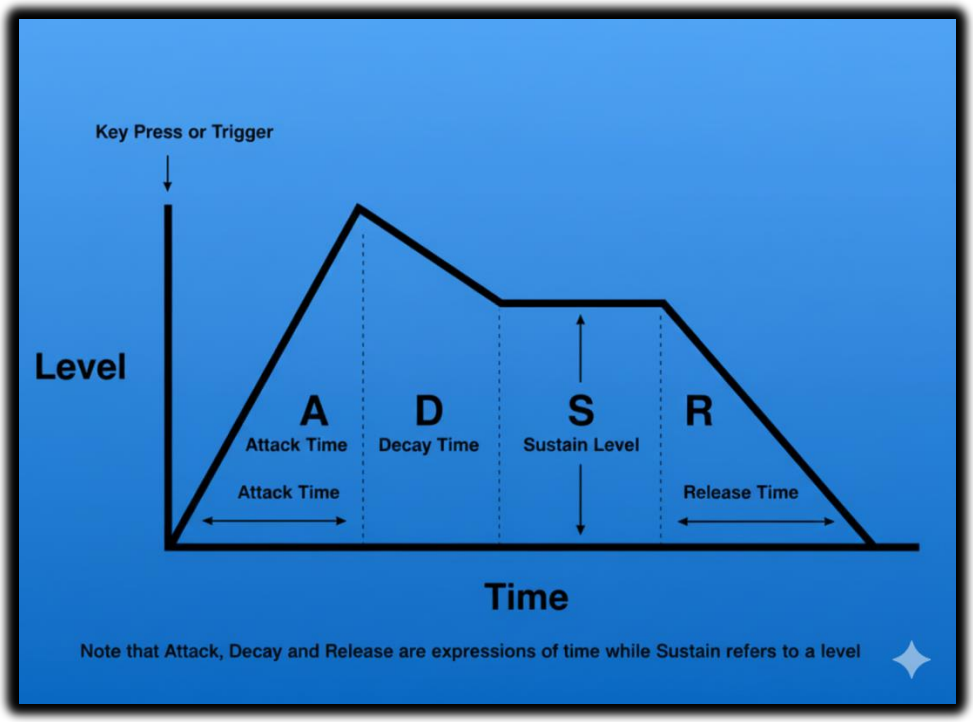
The Importance of the Attack Phase

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:

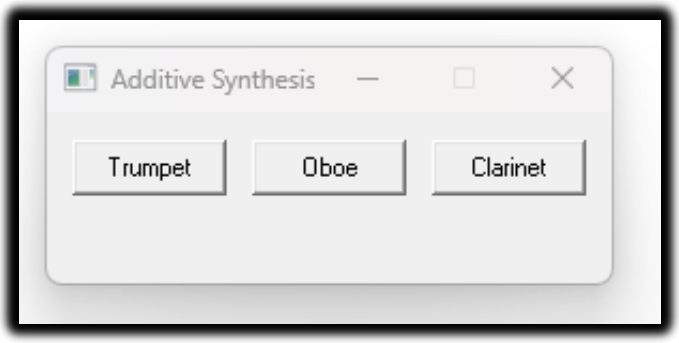
- It reads frequency and amplitude envelopes
- Generates multiple sine waves
- Adds them together
- Sends the result to the sound hardware

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

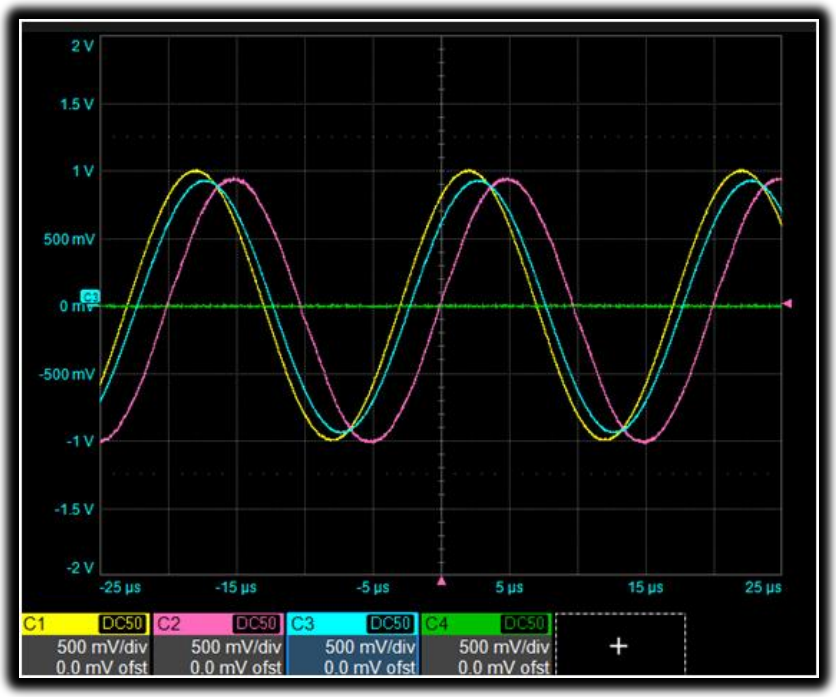


Sine Wave Generation

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.

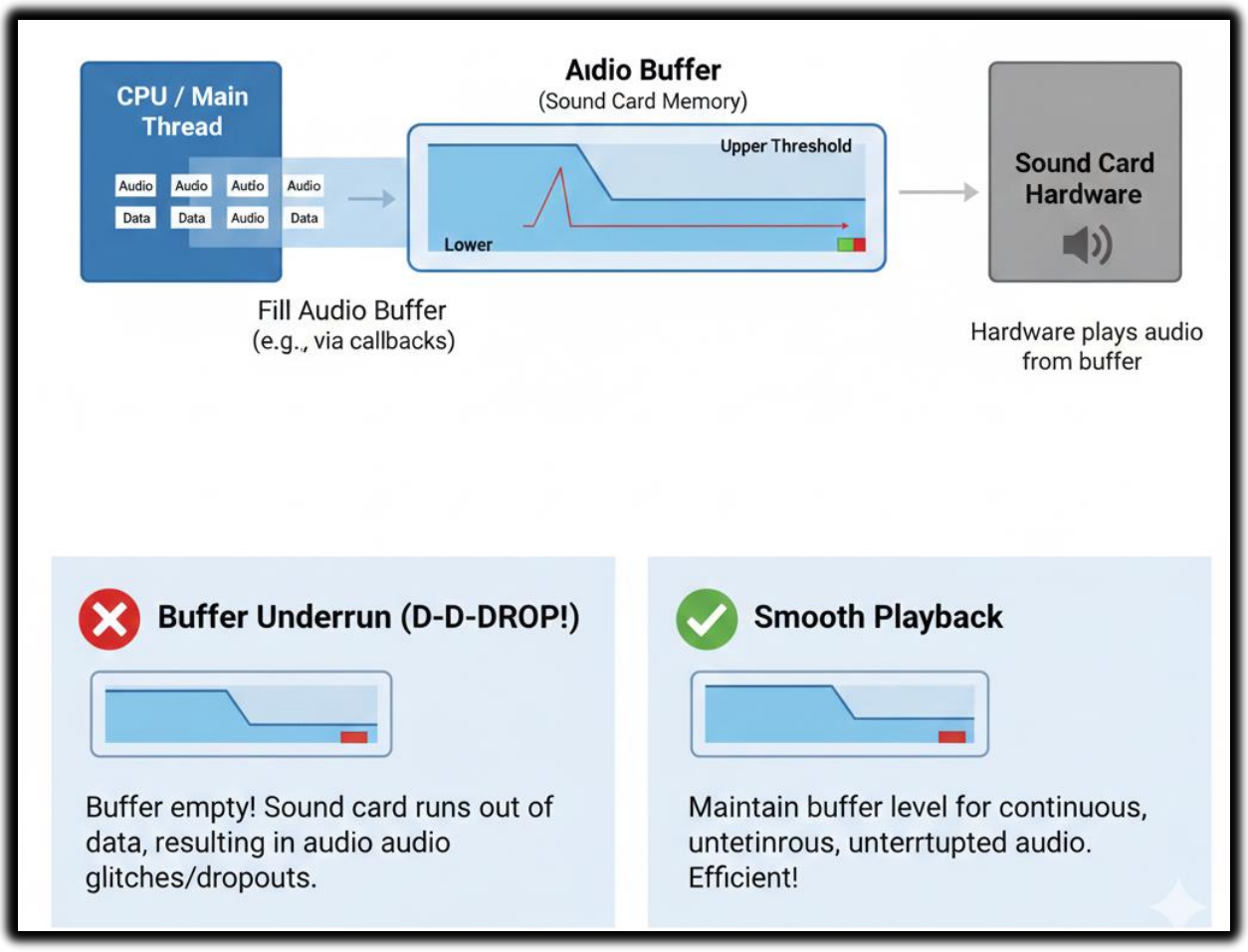
Instead of producing just one sine wave, this function:

- 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

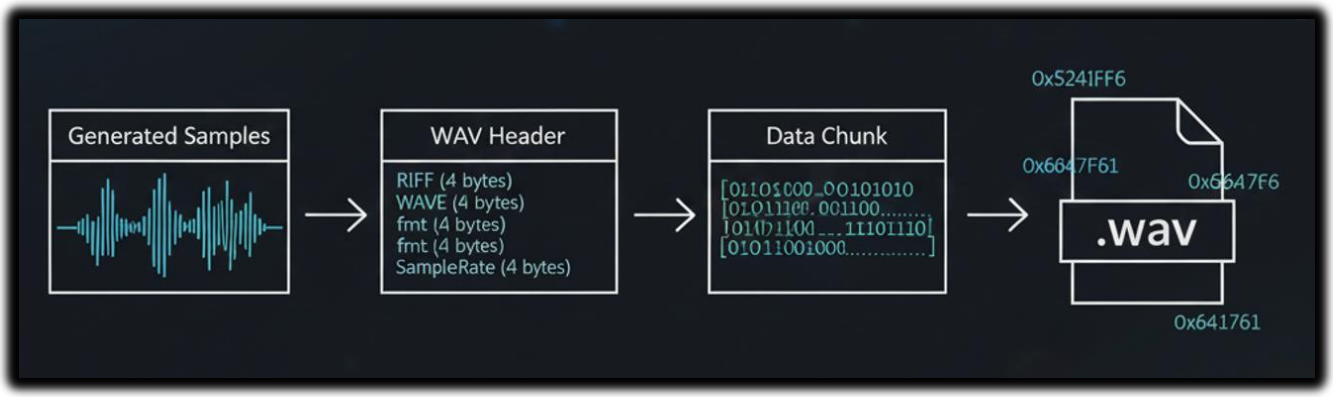
The MakeWaveFile function handles turning generated samples into a real .WAV file.

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:

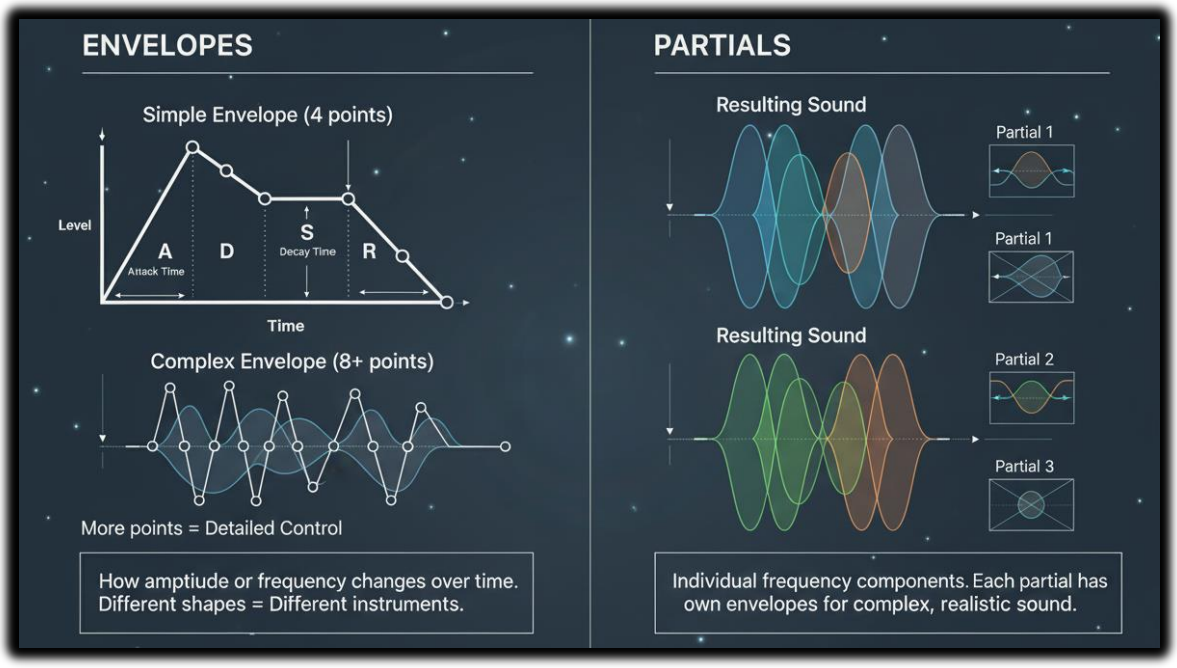
- Total duration
- Number of partials
- Pointer to the PRT array

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.

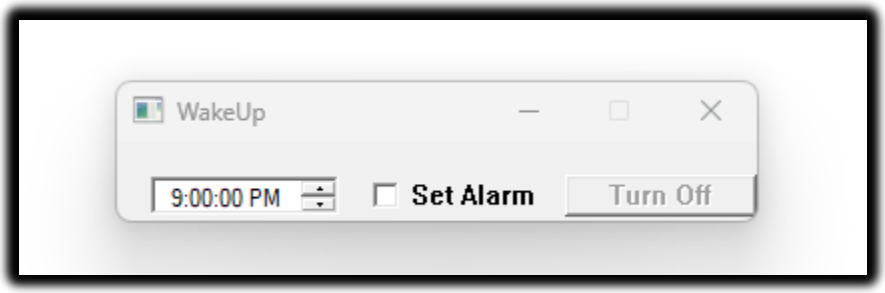
Final Note

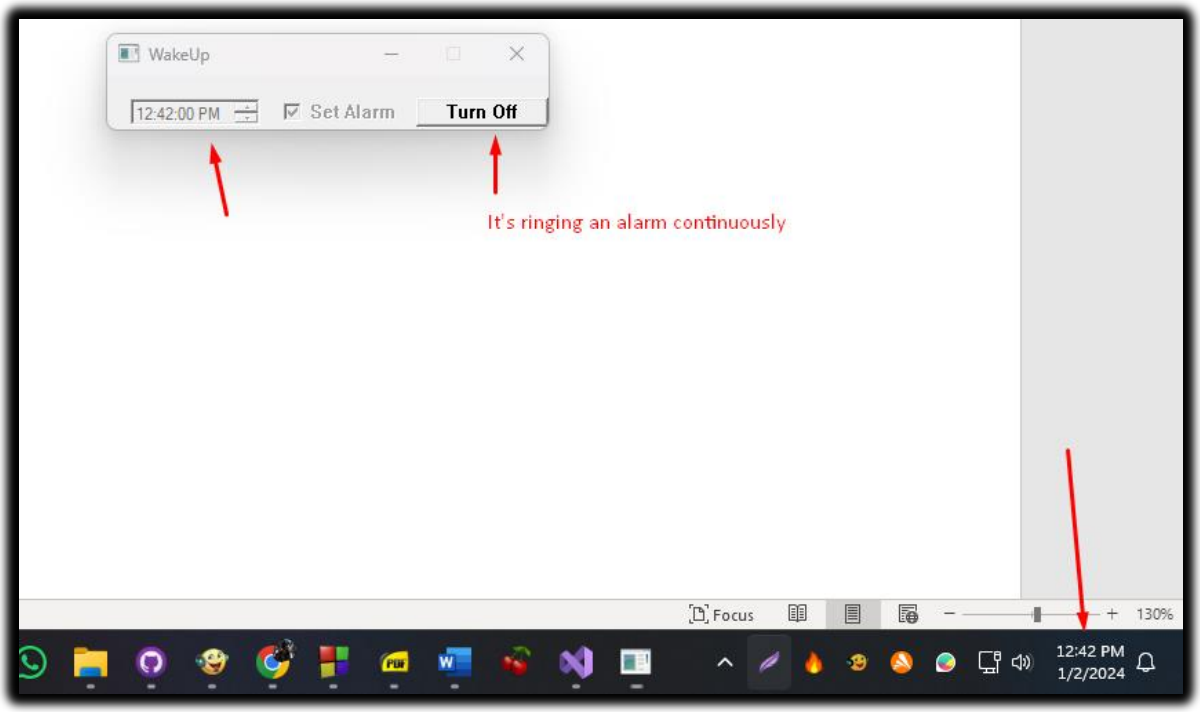
This program is less about flashy UI and more about demonstrating:

- 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

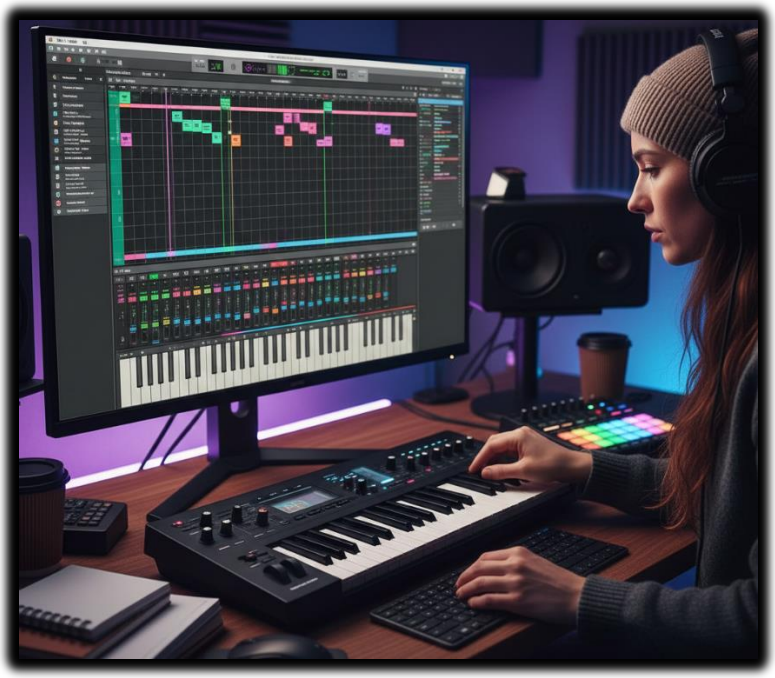
i) Standalone Sequencers

- 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

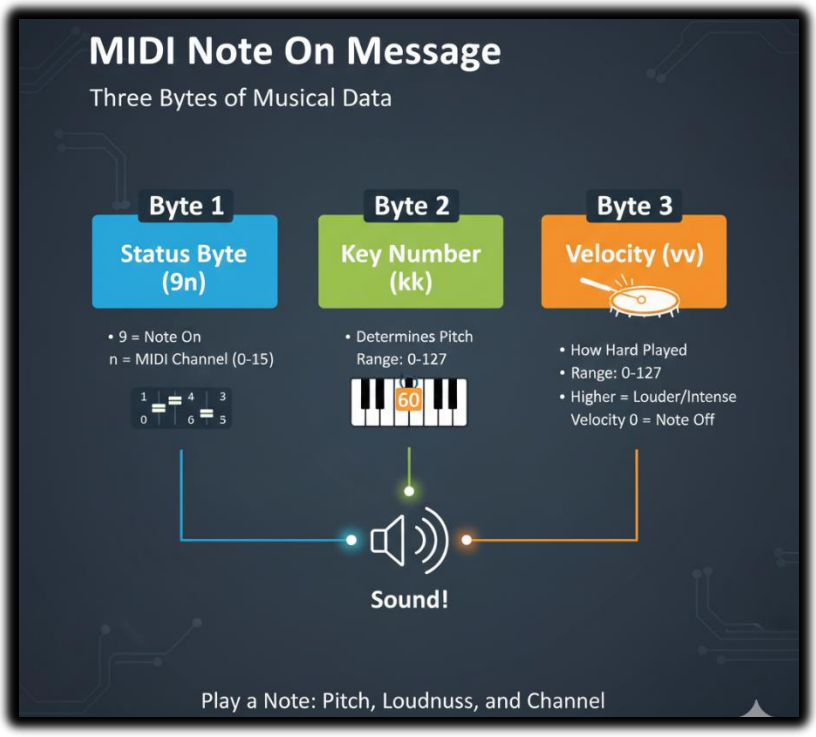
- The status byte acts like a **routing label**.
- It ensures the message reaches the correct instrument.
- Without it, MIDI devices wouldn't know who the message is for.

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)

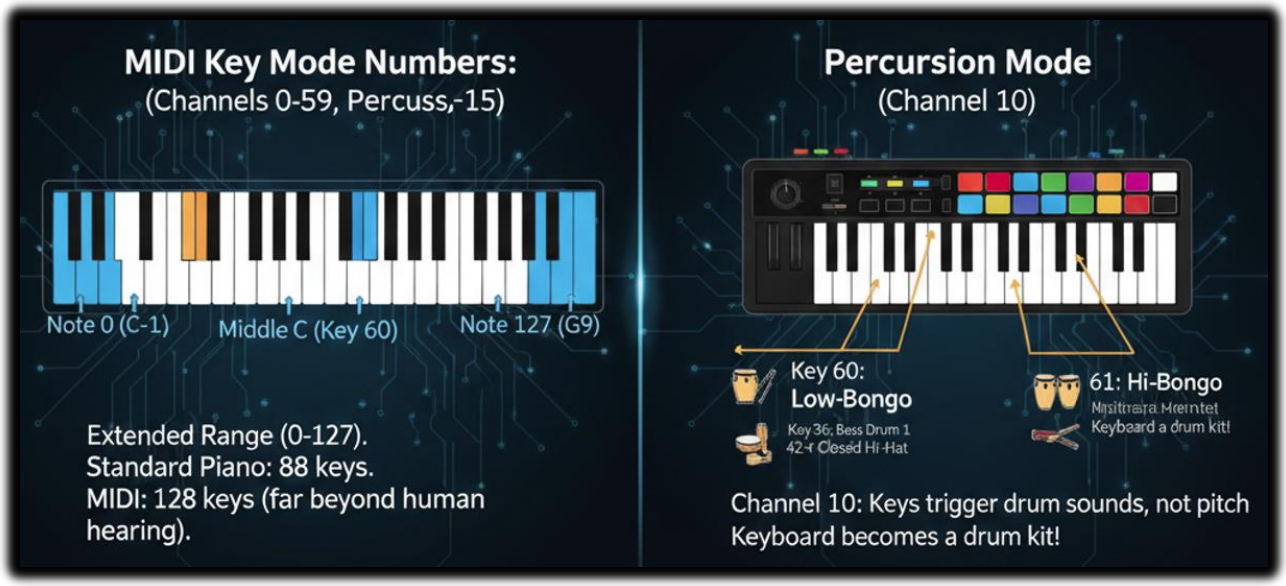
(Note: 'n' represents the Channel Number 0-F)

The Nuances of Expression

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

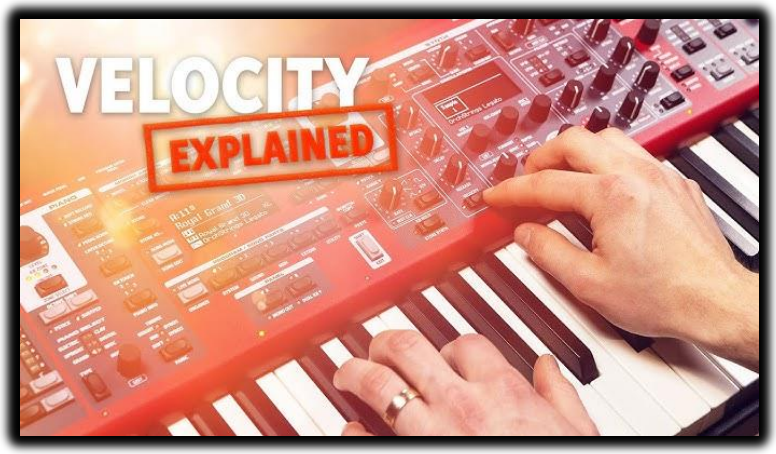
1. Key Numbers (Pitch & Percussion)

- **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.

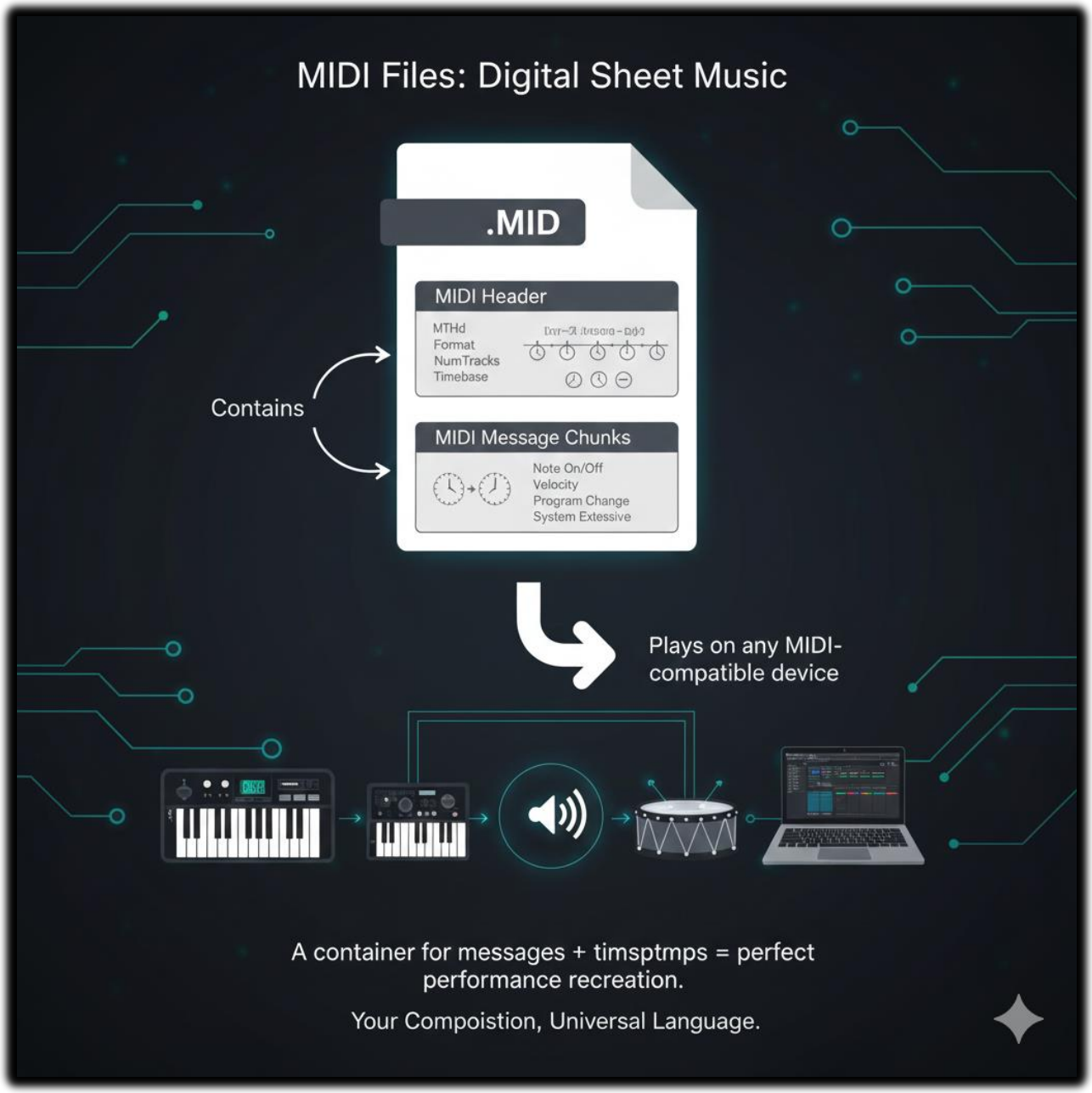
System Exclusive (SysEx): The "Secret Handshake."

- 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.



DELVING INTO MIDI SEQUENCING: ORCHESTRATING MUSIC WITH CODE

Communicating with MIDI Devices: A Low-Level Symphony

While MIDI files offer a way to capture and share musical performances, the true magic of MIDI lies in its ability to control instruments and create music in real-time.

This is where the low-level MIDI API enters the stage, providing a direct line of communication with MIDI devices.

Opening the Stage Door: midiOutOpen

To begin your MIDI journey, you'll first need to establish a connection with the desired MIDI output device, whether it's an internal synthesizer or an external instrument. Here's how you use the midiOutOpen function:

```
#include <windows.h>
#include <mmsystem.h>

// ...

HMIDIOUT hMidiOut;  // Handle for MIDI output device
MMRESULT error;

error = midiOutOpen(&hMidiOut, wDeviceID, NULL, 0, CALLBACK_NULL);
if (error != MMSYSERR_NOERROR) {
    // Handle error, such as device already in use
}
```

After calling midiOutOpen to open the MIDI output device, it is important to handle any potential errors that may occur. Error handling allows you to gracefully respond to issues that may arise, such as the MIDI device already being in use or being unavailable.

In the provided code snippet, the error variable is used to capture the return value of midiOutOpen.

If error is not equal to MMSYSERR_NOERROR, it indicates that an error has occurred during the attempt to open the MIDI output device.

Handling errors typically involves taking appropriate actions based on the specific error code returned. For example, you might display an error message to the user, log the error for debugging purposes, or implement a fallback strategy.

If the error code is MMSYSERR_ALLOCATED, it means that the MIDI output device is already in use by another application or process. In this case, you might want to inform the user that the device is unavailable and allow them to try again later or select a different device if available.

If the error code is MMSYSERR_BADDEVICEID, it indicates that the specified device ID is invalid. This could happen if the device ID provided in wDeviceID is out of range or does not correspond to a valid MIDI output device on the system. You can display an error message to the user informing them of the invalid device ID and allow them to select a different device.

If the error code is MMSYSERR_NODRIVER, it suggests that there is no driver installed for the MIDI output device. You can inform the user that the MIDI device is not supported or that they need to install the appropriate driver for the device to work correctly.

These are just a few examples of how you can handle errors when opening a MIDI output device. The specific actions you take will depend on the requirements of your application and the desired user experience. It's important to provide informative error messages and handle errors gracefully to ensure a smooth user experience when working with MIDI devices.

Sending Musical Messages: midiOutShortMsg

Once the stage is set, you can start sending MIDI messages to create music! Here's how you use the midiOutShortMsg function:

```
DWORD dwMessage = 0x90407F;  // Note On message (channel 1, middle C, velocity 127)
error = midiOutShortMsg(hMidiOut, dwMessage);
if (error != MMSYSERR_NOERROR) {
    // Handle error, such as device not ready
}
```

With the stage set and the MIDI output device open, you can start sending MIDI messages to create music. One way to do this is by using the midiOutShortMsg function. This function allows you to send a MIDI message in the form of a 32-bit DWORD to the MIDI output device.

In the provided code snippet, a DWORD variable named dwMessage is used to store the MIDI message. The message represents a "Note On" event on channel 1, with middle C as the note (0x40) and a velocity of 127 (0x7F). The status byte (0x90) indicates a Note On event on channel 1, and the most significant byte is set to 0.

After composing the MIDI message, you can call midiOutShortMsg to send it to the MIDI output device specified by the hMidiOut handle. If the function call returns an error code other than MMSYSERR_NOERROR, it indicates that there was an issue sending the MIDI message.

Handling errors when sending MIDI messages is essential to ensure the smooth operation of your application. Possible error scenarios include the MIDI device not being ready, the device becoming disconnected, or other system-level issues.

You should handle these errors appropriately, which may involve displaying an error message to the user, logging the error for further investigation, or taking any necessary recovery actions.

It's important to note that the provided code snippet demonstrates sending a "Note On" message, but MIDI offers a wide range of message types for various musical actions. For example, you can explore messages for actions such as changing programs/instruments, controlling pitch bend, altering control parameters, and more. Understanding and utilizing different MIDI messages will allow you to create dynamic and expressive musical experiences.

Once you have finished sending MIDI messages and no longer need the MIDI output device, it is important to close it using the midiOutClose function. This ensures that the resources associated with the device are properly released.

By embracing the low-level capabilities of MIDI and utilizing functions like midiOutShortMsg, you gain the power to compose and conduct your own MIDI symphonies.

You can send precise commands to MIDI devices, shaping and controlling the musical experience to your desired vision.

Whether you're creating simple melodies or orchestrating complex musical arrangements, MIDI provides a flexible and powerful foundation for your musical endeavors. Let the music flow and enjoy the creative possibilities that MIDI offers!

BACHTOCC.C PROGRAM STRUCTURE:

WinMain Function:

The WinMain function serves as the entry point for the Windows application.

It initializes the window class, creates the main window for MIDI playback, and enters the main message loop.

Parameters include hInstance (handle to the current instance of the application), hPrevInstance (historical instance, no longer used), szCmdLine (command-line parameters), and iCmdShow (initial window display state).

WndProc Function:

The WndProc function is the window procedure that processes messages for the main window.

It handles messages such as WM_CREATE for initialization, WM_TIMER for timing events, and WM_DESTROY for window destruction.

In the WM_CREATE case, it opens the MIDIMAPPER device and sends Program Change messages to set the instrument to "Church Organ."

The function also manages the polyphonic playback logic, note sequencing, and MIDI messages.

MidiOutMessage Function:

The MidiOutMessage function encapsulates the creation and sending of MIDI messages.

It takes parameters such as the MIDI output handle (hMidi), MIDI status, channel, data1, and data2.

It constructs a DWORD message using bitwise operations and sends the MIDI message using midiOutShortMsg.

MIDI FUNCTIONALITY:

MidiOutMessage Function:

This function is crucial for generating MIDI messages.

It takes the specified parameters and constructs a DWORD value representing a MIDI message using bitwise operations.

The resulting message is sent to the MIDI output device using the midiOutShortMsg function.

WM_CREATE Case in WndProc:

In the WM_CREATE case, the program initializes MIDI functionality.

It opens the MIDIMAPPER device using midiOutOpen, which establishes communication with the MIDI output device.

Program Change messages (0xC0) are then sent to set the instrument to "Church Organ" on both channel 0 and channel 12.

These components collectively form the program's structure and MIDI functionality. The program creates a window, sets up MIDI communication, and initiates playback of the first bar of Bach's Toccata in D Minor using polyphonic MIDI sequencing. The use of Program Change messages allows the selection of a specific instrument for playback, enhancing the musical experience.

NOTE SEQUENCING:

Static Array (noteseq):

The program incorporates a static array named noteseq to represent the note sequence of the first bar of Bach's Toccata in D Minor.

Each entry in noteseq holds information about the note duration (iDur) and two simultaneous notes (iNote[0] and iNote[1]).

This structured array serves as a musical score, defining when and which notes should be played during the polyphonic playback.

Duration and Simultaneous Notes:

- iDur represents the duration of the note in milliseconds, indicating how long a note should be held.
- iNote[0] and iNote[1] contain the MIDI note numbers of two simultaneous notes to create a polyphonic effect.
- The inclusion of simultaneous notes allows for the representation of chords or harmonies in the musical sequence.

POLYPHONIC PLAYBACK:

Two-Note Polyphony:

The program employs a polyphonic playback mechanism that supports a two-note polyphony.

This means that the musical sequence can include two simultaneous notes, enhancing the richness of the generated sound.

MIDI Messages for Playback:

The polyphonic playback is achieved through the generation of MIDI Note On and Note Off messages.

When a note is to be played (Note On), the program sends MIDI messages with the appropriate note numbers and velocity.

Conversely, when a note is to be stopped (Note Off), corresponding MIDI messages are sent with a velocity of 0, indicating the release of the note.

Timing with WM_TIMER Case:

The WM_TIMER case in the WndProc function is crucial for managing the timing of note playback.

It handles the progression through the noteseq array, triggering the Note On and Note Off messages at specific intervals based on the defined note durations.

The use of a timer ensures accurate timing, creating a seamless musical experience during playback.

Dynamic Note Sequencing:

The program dynamically advances through the noteseq array, playing the specified notes and maintaining the polyphonic structure.

As each note's duration expires, the program proceeds to the next set of notes in the sequence, creating a fluid and dynamic musical output.

In summary, the note sequencing and polyphonic playback mechanisms in the program work cohesively to interpret and translate the musical score (noteseq array) into a real-time, polyphonic MIDI performance. The use of simultaneous notes and accurate timing enhances the overall musicality of the generated sound during playback.

TIMER HANDLING:

SetTimer Function:

The SetTimer function is employed to establish a timer, which triggers the WM_TIMER case at regular intervals.

In this program, it ensures the advancement through the note sequence, facilitating polyphonic playback.

The timer is set with a specified interval, and the WM_TIMER case in the WndProc function handles the events triggered by this timer.

KillTimer Function:

The KillTimer function is used to stop the timer when the note sequence is complete.

After the timer is killed, the program proceeds to close the MIDI output and exit gracefully.

This mechanism ensures that the program concludes its musical playback and cleans up resources appropriately.

WINDOW PROCEDURE:

WndProc Function:

The WndProc function serves as the window procedure, handling various Windows messages to control program behavior.

Messages such as WM_CREATE, WM_TIMER, and WM_DESTROY are managed within this function.

WM_CREATE Case:

In the WM_CREATE case, the program initializes the MIDI output device.

It opens the MIDIMAPPER device, sends Program Change messages to set the instrument to "Church Organ," and establishes the initial state for MIDI playback.

Additionally, it sets the timer to initiate polyphonic note sequencing.

WM_TIMER Case:

The WM_TIMER case is crucial for managing the timing of note playback.

It handles the progression through the note sequence, triggering MIDI Note On and Note Off messages at specific intervals based on the defined note durations.

The dynamic nature of the WM_TIMER case ensures accurate timing for a seamless musical experience during playback.

WM_DESTROY Case:

In the WM_DESTROY case, the program resets and closes the MIDI output.

The midiOutReset function resets the MIDI output, and midiOutClose closes the MIDI output device.

Finally, the program posts a quit message to exit the application gracefully.

Window Creation:

The program creates a window with a specified title and dimensions using the CreateWindow function.

The window is set to play the Bach Toccata when it is created, initiating the MIDI playback.

Window Destruction:

Upon window destruction (handled in WM_DESTROY case), the program resets the MIDI output using midiOutReset and closes the MIDI output device with midiOutClose.

Proper cleanup ensures that resources are released, providing a smooth exit for the program.

BACCTOCC Program.mp4

In summary, the program seamlessly combines Windows GUI functionality with MIDI sequencing. It creates a window for MIDI playback, handles note sequencing with precise timing using timers, and ensures proper cleanup upon program termination. The integration of Windows messages and MIDI functionality results in a cohesive and responsive musical experience.

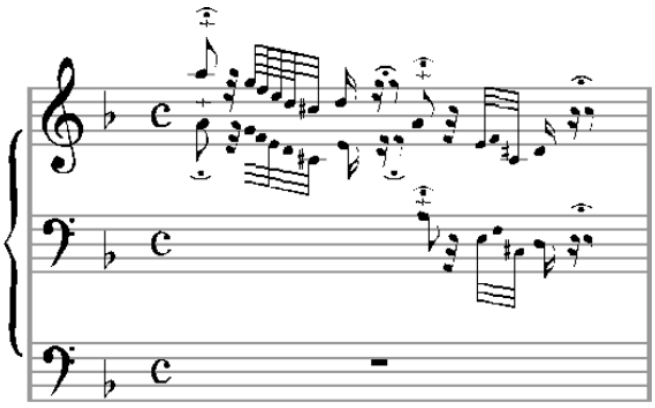


Figure 22–11. *The first measure of Bach's Toccata and Fugue in D Minor.*

Limitations of Windows Timer:

- BACHTOCC demonstrates playing music using the Windows timer, but the approach has limitations.
- The Windows timer, based on the system clock, lacks the resolution required for accurate music reproduction.
- Asynchronous and timing issues may occur, affecting the quality of music playback.

Alternative Timer Functions:

- Windows provides supplementary timer functions with the prefix "time" for improved resolution (as low as 1 millisecond).
- These functions can be utilized for accurate timing when working with low-level MIDI output functions.

KBMIDI PROGRAM IN DEPTH

MIDI Communication:

1. midiOutOpen, midiOutShortMsg, and midiOutClose:

midiOutOpen: This function is used to open a connection to a MIDI output device. It requires parameters like the device identifier, callback function, and instance data. Handling the device identification and ensuring proper initialization is critical. Error checking is vital to address potential issues in device availability or opening.

midiOutShortMsg: Responsible for sending short MIDI messages to the specified MIDI output device. Understanding the structure of MIDI messages is essential, as this function is crucial for note on/off, control changes, and other real-time adjustments. Efficient management of MIDI channels, instruments, and control messages is imperative for accurate and expressive musical output.

midiOutClose: Closes the MIDI output device opened with midiOutOpen. Proper cleanup, resource release, and error handling are essential to ensure the graceful termination of the MIDI communication. Failing to close the device properly may lead to resource leaks and impact system stability.

2. Managing MIDI Channels, Instruments, and Control Messages:

MIDI Channels: MIDI communication involves 16 channels, each capable of carrying independent musical data. The program must manage channel assignments for different instruments and ensure that the right messages are sent to the correct channel.

Instruments (Voices): Assigning and handling different instruments or voices require a deep understanding of the General MIDI (GM) standard or any custom instrument mapping used by the program. This includes managing program change messages to switch between instruments dynamically.

Control Messages: Beyond note on/off messages, MIDI offers control messages for parameters like pitch bend, modulation, and expression. Proper handling of these messages is crucial for achieving nuanced and expressive musical output.

Window Procedure (WndProc):

1. Handling Keyboard Input:

Key Press/Release Events: WndProc must efficiently differentiate between key press and release events to trigger the corresponding actions. This involves tracking the state of each key and updating it dynamically.

Mapping Keyboard Input to MIDI Notes: The program likely includes a mapping mechanism to associate each key with a specific MIDI note. This mapping should consider factors like octaves and potentially handle different key layouts or configurations.

2. Menu Commands:

Dynamic Menu Creation: Depending on the available MIDI devices, the program dynamically creates menus. This involves querying the system for MIDI devices, obtaining their capabilities, and generating corresponding menu items. Error handling is crucial for situations where MIDI devices may not be available or accessible.

Menu Command Handling: WndProc processes menu commands triggered by user interactions. This includes actions like opening/closing MIDI devices, changing channels, and selecting instruments. Proper synchronization with MIDI communication functions ensures a seamless user experience.

3. Scrollbar Adjustments:

Horizontal and Vertical Scroll Bars: The program likely uses scroll bars to control parameters such as velocity and pitch bend. WndProc must respond to scroll bar messages, updating the associated parameters and reflecting these changes in real-time.

Visual Feedback: Inverting the color of keys when pressed provides visual feedback. Managing this visual representation involves handling WM_PAINT messages and updating the window's appearance dynamically.

In summary, both MIDI communication and the Window Procedure are intricate components requiring a deep understanding of real-time musical data processing, user interface interactions, and system-level resource management. The effective coordination of these elements is essential for creating a responsive and feature-rich MIDI synthesizer application.

MENU CREATION AND HANDLING:

1. Dynamic Menu Creation:

Enumerating MIDI Devices: The program likely uses MIDI API functions to enumerate available MIDI devices. This involves querying the system for MIDI input and output devices, obtaining device information, and dynamically creating menu items based on this information.

Device Capabilities: Each MIDI device may have unique capabilities. The program must extract and interpret these capabilities to provide meaningful menu options. For instance, displaying the available channels, instruments, or other device-specific settings in the menu.

Dynamic Menu Update: The program must dynamically update the menu whenever the number or configuration of MIDI devices changes. This includes scenarios where devices are added or removed during runtime.

2. Responding to Menu Commands:

Open/Close MIDI Devices: WndProc responds to menu commands related to opening and closing MIDI devices. Proper error handling is essential to address situations where a device cannot be opened or closed successfully, ensuring the stability of the MIDI communication.

Change Channels and Select Instruments: The program allows users to change MIDI channels and select instruments through menu commands. Coordination with MIDI communication functions is crucial to ensure that these changes are reflected in real-time, affecting the program's musical output.

KEYBOARD INPUT PROCESSING:

1. Mapping Keyboard Input to MIDI Notes and Octaves:

Key to Note Mapping: WndProc processes keyboard input, mapping each key to a corresponding MIDI note. This involves creating a mapping table that associates keys with specific note values, considering factors like the starting octave and potential octave adjustments.

Octave Adjustments: The program likely allows users to adjust the octave using modifiers like Shift and Ctrl. WndProc must interpret these modifier states and dynamically update the mapping, allowing users to play across multiple octaves with ease.

Visual Feedback: In addition to MIDI output, the program may provide visual feedback for pressed keys, enhancing the user experience. This involves updating the appearance of keys in the graphical user interface to reflect the current state of each key, providing a responsive and immersive interaction.

2. Managing Key States:

Key Press and Release Events: WndProc efficiently manages key press and release events, maintaining the state of each key. This involves handling the WM_KEYDOWN and WM_KEYUP messages, updating the internal key state, and triggering the corresponding MIDI note events.

Modifiers (Shift, Ctrl): Recognizing and interpreting modifier keys is crucial for implementing advanced features like octave adjustments. WndProc tracks the state of these modifiers, dynamically adjusting the mapping table to accommodate changes in octave settings during live play.

In summary, effective menu creation and keyboard input processing are integral to the overall functionality and user experience of a MIDI synthesizer program. These components require a combination of system-level interaction, real-time data processing, and dynamic user interface updates to deliver a seamless and expressive musical environment.

SCROLL BAR CONTROLS:

1. Incorporating Horizontal and Vertical Scroll Bars:

Parameter Control: The program uses scroll bars to allow users to control parameters like velocity and pitch bend. Horizontal and vertical scroll bars likely correspond to different parameters, providing a visual and interactive way for users to adjust these settings dynamically.

Mapping to MIDI Parameters: Each scroll bar's position is mapped to specific MIDI parameters. For example, the vertical scroll bar might control pitch bend, while the horizontal scroll bar adjusts velocity. Ensuring a smooth and intuitive mapping is crucial for user-friendly parameter manipulation.

2. Handling Scroll Bar Messages:

Message Processing: WndProc processes messages related to scroll bar interaction, such as SB_LINEUP, SB_LINEDOWN, SB_THUMBPOSITION, etc. These messages signify user actions like scrolling, dragging the thumb, or clicking on the scroll bar arrows.

Real-time Updates: The program must update MIDI parameters in real-time based on scroll bar interactions. This involves translating the scroll bar positions into meaningful changes in velocity and pitch bend, ensuring that the musical output reflects the user's adjustments accurately.

GRAPHICS AND DRAWING:

1. Drawing Individual Keys on the Window:

Key Positioning and Dimensions: The program dynamically calculates and positions individual keys on the window, considering factors like key size, spacing, and overall layout. This is crucial for creating an accurate and visually appealing representation of the musical keyboard.

Labeling Keys: Each key is labeled based on its MIDI note value. This labeling provides users with a clear visual reference, aiding in navigation and musical expression. Labels may include note names (C, D, E, etc.) and octave numbers.

2. Inverting Color of Keys When Pressed:

Visual Feedback for Key Press: When a key is pressed, the program inverts the color of the corresponding graphical representation. This visual feedback enhances the user experience, providing a responsive and intuitive connection between the virtual interface and the user's actions.

Dynamic Color Handling: The program dynamically manages the color state of keys, ensuring that the inversion occurs only when a key is actively pressed. This involves handling key press and release events, updating the graphical representation accordingly.

User Engagement: The visual feedback not only serves a functional purpose but also contributes to the overall engagement and immersion of the user in the virtual musical environment. It mimics the tactile response of a physical keyboard, enhancing the sense of interaction and control.

In summary, the incorporation of scroll bar controls and the thoughtful design of graphics and drawing elements are essential aspects of your MIDI program. These components contribute to the program's usability, real-time responsiveness, and visual appeal, creating a rich and enjoyable experience for users interacting with virtual musical instruments.

GRAPHICS AND DRAWING:

1. Drawing Individual Keys on the Window:

Position Calculation: The program employs precise algorithms to calculate the positions of individual keys dynamically. This involves considering factors like the total number of keys, their dimensions, and the overall layout of the virtual keyboard. The goal is to create an accurate and visually pleasing representation of the musical keyboard within the application window.

Dimensional Considerations: Each key is drawn with careful attention to its dimensions, ensuring that they are proportional and visually coherent. This contributes to the realism and aesthetics of the virtual instrument, providing users with a visually intuitive representation that mirrors the physical attributes of a traditional musical keyboard.

Labeling for Reference: Keys are labeled based on their MIDI note values, incorporating note names and octave numbers. This labeling enhances user understanding and navigation, allowing musicians to easily identify and locate specific notes on the virtual keyboard.

2. Inverting the Color of Keys When Pressed:

Dynamic Color Handling: The program dynamically manages the color state of keys in response to user interactions. When a key is pressed, its color is inverted, providing immediate and intuitive visual feedback. This feature mimics the tactile response of physical keys, enhancing the user's sense of connection and control.

Visual Feedback for User Engagement: The color inversion serves not only a functional purpose but also contributes to user engagement. It creates a responsive and interactive environment, reinforcing the connection between the user's actions and the virtual instrument's visual representation. This feedback is crucial for musicians who rely on visual cues during performance and composition.

RESOURCE MANAGEMENT:

1. Properly Opening and Closing MIDI Devices:

MIDI API Functions: The program utilizes MIDI API functions, such as midiOutOpen and midiOutClose, to establish and terminate connections with MIDI devices. Proper handling of these functions ensures the reliable and efficient communication between the software and external hardware.

Error Handling: Robust error-handling mechanisms are implemented to address potential issues during device opening and closing. This includes checking return values, diagnosing errors, and providing informative messages to users in case of failures. This proactive approach enhances the overall stability and reliability of the MIDI communication.

2. Managing Resources and Handling Potential Errors:

Resource Allocation: The program efficiently manages system resources, such as memory and processing power, to ensure optimal performance. This involves judicious allocation and deallocation of resources, preventing memory leaks and optimizing the overall efficiency of the application.

Error Resilience: Robust error-handling strategies are integrated throughout the code to anticipate and address potential errors gracefully. This includes scenarios such as device

unavailability, resource exhaustion, or unexpected runtime issues. By handling errors systematically, the program enhances its resilience and user experience.

CODE ORGANIZATION:

1. Using Structures and Constants for Organization:

Structured Data Representation: The program employs structures to organize MIDI instrument families, instruments, and key mappings. This structured approach enhances code readability and maintainability by encapsulating related data into cohesive units. Each structure likely contains essential information such as instrument names, MIDI channels, and key assignments.

Constants for Configuration: Constants are used to define fixed values that remain unchanged during program execution. They are employed to represent parameters like instrument families, ensuring consistency and facilitating easy updates. This modular organization improves code comprehensibility and simplifies future modifications.

Enhanced Modularity: By utilizing structures and constants, the code achieves a modular and organized structure. This design choice enhances scalability, allowing for the seamless addition of new instrument families or modifications to existing ones. It reflects a thoughtful approach to code organization, promoting clarity and ease of maintenance.

ERROR HANDLING:

Limited Error Handling:

Critical Error Messaging: The current error-handling approach employs basic message boxes for critical errors. While effective for conveying essential information, it's crucial to consider expanding error handling to cover a broader range of scenarios. This can include non-critical errors, providing users with informative messages to aid troubleshooting and ensuring a more user-friendly experience.

Logging Mechanism: Implementing a logging mechanism can enhance error tracking and debugging. Logging critical events and errors to a file allows developers to analyze issues post-execution, facilitating quicker identification and resolution. This proactive approach improves the overall robustness of the application.

UNDERSTANDING MIDI MESSAGES:

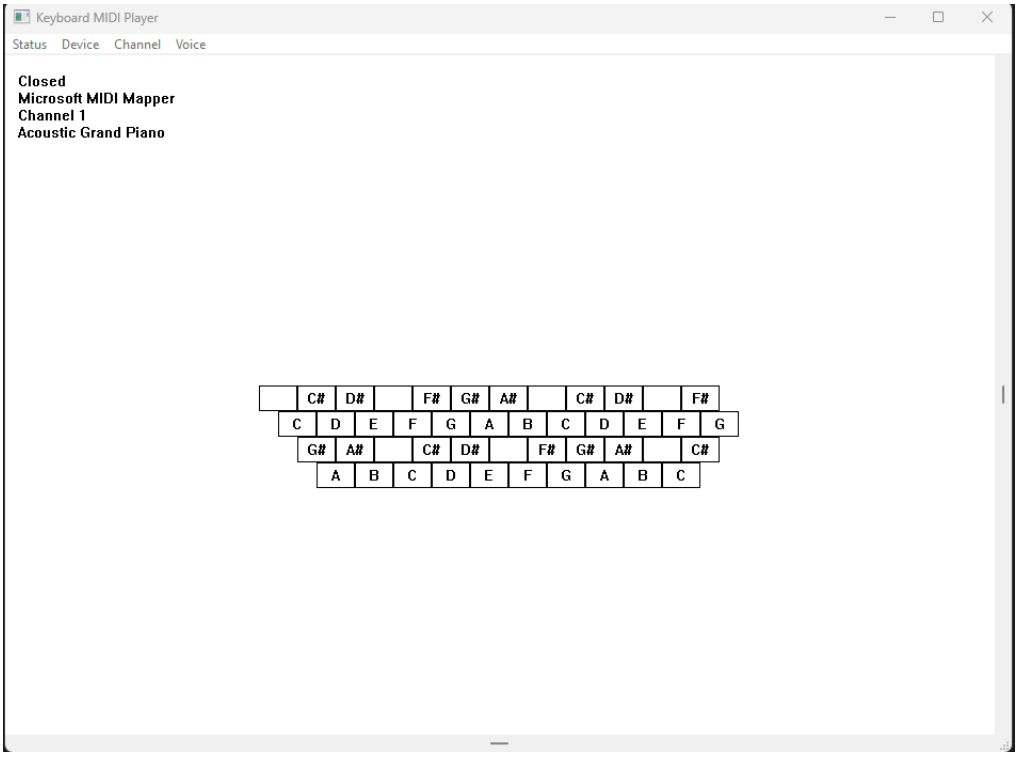
Knowledge of MIDI Message Formats:

Message Structure Familiarity: The code demonstrates a profound understanding of MIDI message formats and their interpretation. This is evident in the effective use of MIDI API functions, such as midiOutShortMsg, which is pivotal for sending MIDI messages to devices. The correct construction and parsing of MIDI messages indicate a high level of proficiency in MIDI communication.

Channel and Control Handling: Managing MIDI channels, instruments, and control messages involves intricate knowledge of MIDI message structures. The program showcases competence in handling complex MIDI messages, ensuring precise control over parameters like instrument selection and channel configuration.

Closing Thoughts:

In conclusion, the provided code serves as an exemplary demonstration of expertise in WinAPI development, MIDI communication, and user interface design tailored for a MIDI synthesizer application. It stands as a robust foundation for a functional and interactive program, showcasing meticulous attention to detail in various aspects.



The code's strengths lie in its:

- Efficient MIDI Communication: Leveraging MIDI API functions for device interaction and demonstrating a nuanced understanding of MIDI message formats.
- User Interface Design: Incorporating dynamic elements like dynamic menu creation, keyboard input processing, scroll bar controls, and visually engaging graphics.
- Code Organization: Utilizing structures and constants for a modular and organized codebase, enhancing readability and maintainability.
- While there is room for refining error handling and potentially expanding features, the code overall represents a commendable achievement in WinAPI development, catering to the intricate demands of MIDI synthesis and user interaction. It stands as a testament to the developer's proficiency and creativity in crafting a compelling musical application.

More Explanation:

Keyboard Mapping:

- The program maps the computer keyboard to traditional piano or organ keys.
- The Z key plays an A at 110 Hz, and the keyboard spans 3 octaves from the bottom row to the top two rows.

- The Ctrl key drops the entire range by 1 octave, while the Shift key raises it by 1 octave, providing an effective range of 5 octaves.

MIDI Device Management:

- To hear sound, users need to select "Open" from the Status menu to open a MIDI output device.
- Pressing a key sends a MIDI Note On message, and releasing the key generates a Note Off message.
- "Close" from the Status menu allows users to close the MIDI device without terminating the program.
- The "Device" menu lists installed MIDI output devices, including MIDI Out ports and the MIDI Mapper device.

MIDI Channel and Voice Selection:

- The "Channel" menu lets users select a MIDI channel from 1 through 16, with channel 1 as the default.
- The "Voice" menu offers a selection of 128 instrument voices from the General MIDI specification, divided into 16 families with 8 instruments each.
- These voices, known as melodic voices, correspond to different pitches based on MIDI key numbers.

Percussion Instruments:

- The "Channel" menu, when set to channel 10, allows users to play nonmelodic percussion instruments.
- Selecting the first instrument voice (Acoustic Grand Piano) from the "Voice" menu enables different percussion sounds for each key.

Scroll Bars:

- Horizontal and vertical scroll bars control parameters like note velocity and pitch bend.
- The horizontal scroll bar, due to the non-velocity-sensitive nature of the PC keyboard, adjusts the volume of played notes.

Vertical Scroll Bar and Pitch Bend:

- The vertical scroll bar in KBMIDI is responsible for generating a MIDI message known as "Pitch Bend."
- Users can press down one or more keys and manipulate the vertical scroll bar thumb with the mouse.

- Raising the scroll bar thumb increases the frequency of the note, while lowering it decreases the frequency. Releasing the scroll bar returns the pitch to normal.

Considerations for Scroll Bars:

- Manipulating scroll bars interrupts the program's message loop for keyboard messages.
- If a key is pressed and the scroll bar is manipulated before releasing the key, the note will continue to sound.
- It's advised not to press or release keys during scroll bar manipulation, and caution is needed when interacting with menus.

Handling Stuck Notes:

- If notes get "stuck" and continue to sound after release, pressing the Esc key sends 16 "All Notes Off" messages to stop the sounds.

Menu and Resource Handling:

- KBMIDI creates its menu from scratch without a resource script.
- Device names are obtained from the midiOutGetDevCaps function, and instrument voice families and names are stored in a program data structure.

Pitch Bend Message:

- KBMIDI has short functions for simplifying MIDI messages, including the Pitch Bend message.
- The Pitch Bend message uses two 7-bit values to form a 14-bit pitch-bend level, where values between 0 and 0x1FFF lower the pitch, and values between 0x2001 and 0x3FFF raise the pitch.

MidiSetPatch and Device Changes:

- When selecting "Open" from the Status menu, KBMIDI calls midiOutOpen for the selected device and, if successful, calls its MidiSetPatch function.
- Changing the device requires closing the previous device if necessary and reopening the new device, with a call to MidiSetPatch when changing the MIDI device, channel, or instrument voice.

WM_KEYUP and WM_KEYDOWN Messages:

- KBMIDI processes WM_KEYUP and WM_KEYDOWN messages to turn notes on and off.
- A data structure maps keyboard scan codes to octaves and notes, used for both MIDI note generation and drawing keys on the window.

Scroll Bar Processing:

- Horizontal scroll bar processing involves storing the new velocity level and setting the new scroll bar position.
- Vertical scroll bar processing for pitch bend involves handling SB_THUMBTRACK and SB_THUMBPOSITION commands, setting the scroll bar position to its middle level and calling MidiPitchBend with a value of 8192.

DRUM.C PROGRAM

Header File (DRUMTIME.H):

NUM_PERC: This constant represents the number of percussion instruments in the drum machine, and it is set to 47. This value is crucial for defining the size of arrays or data structures related to percussion instruments.

WM_USER_NOTIFY, WM_USER_FINISHED, WM_USER_ERROR: These are custom Windows messages. Custom messages allow communication between different parts of the program, such as notifying the user interface about events in the MIDI sequence.

DRUM structure: This structure encapsulates various parameters essential for the drum machine. It includes fields for tempo, velocity, the number of beats, and arrays storing

percussion sequences. The structure serves as a container for organizing and passing these parameters within the program.

Functions:

DrumSetParams: This function takes a pointer to a DRUM structure and sets its parameters based on the provided values. It is responsible for initializing or updating the drum machine configuration.

DrumBeginSequence: Initiates the MIDI sequence for the drum machine. This function likely involves setting up MIDI devices and starting the playback of the programmed drum sequence.

DrumEndSequence: Terminates the MIDI sequence for the drum machine. This function could involve stopping MIDI playback and performing cleanup tasks.

Main Program (DRUM.C):

The program includes various standard Windows and C library headers, indicating its reliance on fundamental functionalities provided by the operating system and the C language.

The inclusion of a resource file header suggests that the program utilizes external resources (possibly graphical or textual) that are managed separately.

Global Variables:

- bNeedSave: This flag serves as a boolean indicator, likely tracking whether there are unsaved changes in the drum sequence. It influences user prompts and decisions related to saving modifications.
- drum: An instance of the DRUM structure, holding parameters and sequences for the drum machine. This variable likely represents the state of the drum machine throughout the program.
- hMenu: This variable holds the handle to the program's menu. Menu handles are essential for manipulating and updating menu items during runtime.
- iTempo: Represents the current tempo setting of the drum machine. It is a numerical value determining the speed of the drum sequence.
- szFileName, szTitleName: These strings store the current file name and title, respectively. They are essential for tracking the state of the loaded or saved file.
- cxChar, cyChar: These variables represent the width and height of characters in dialog units. They are likely used for calculating and specifying the layout and dimensions of GUI elements.
- hInst: The handle to the program instance is crucial for interacting with the Windows operating system, especially during window creation and message processing.

These global variables collectively maintain the program's state, user interface elements, and information related to the drum machine's configuration and sequence. They play pivotal roles in controlling the program's behavior and ensuring proper interaction with the user and the operating system.

WinMain Function:

The WinMain function serves as the program's entry point, orchestrating the initialization, window creation, and message processing. It follows a standard structure for Windows applications. Upon invocation, it registers the window class, creates the main window, and enters the message loop, where it awaits and dispatches messages.

The initialization section defines the window class (WNDCLASS structure) with specific attributes, such as window procedure (WndProc), cursor, background brush, and icon. The window class is then registered with the system using RegisterClass.

Subsequently, the main window is created using CreateWindow, and its handle is stored in hwnd. The window is then displayed using ShowWindow, and the message loop is initiated with UpdateWindow.

This function encapsulates the essential setup steps for a typical Windows application, ensuring that the program is ready to receive and respond to user interactions through the window.

WndProc Function:

The WndProc function serves as the window procedure, handling various window messages to manage the program's behavior.

It encompasses a switch statement that processes messages such as WM_CREATE, WM_COMMAND, WM_LBUTTONDOWN, WM_HSCROLL, WM_VSCROLL, WM_PAINT, WM_CLOSE, and others.

In response to WM_COMMAND, menu commands trigger specific actions, such as creating a new file, opening an existing file, saving, and handling sequence playback.

Mouse events (WM_LBUTTONDOWN) and scroll bar adjustments (WM_HSCROLL, WM_VSCROLL) are managed to facilitate user interaction, such as setting the tempo and velocity or selecting beats in the drum sequence. Additionally, WM_PAINT is responsible for rendering the graphical elements on the window.

Custom messages like WM_USER_NOTIFY and WM_USER_FINISHED are used for internal communication, possibly related to MIDI sequence notifications and error handling.

The WndProc function acts as the central hub for interpreting and responding to various events, ensuring the program's responsiveness and maintaining a coherent user interface.

AboutProc Function:

The AboutProc function is a dialog procedure specifically designed for the About box.

It handles messages related to the initialization of the dialog (WM_INITDIALOG) and user commands, such as clicking the OK button (IDOK).

The function returns TRUE for messages it processes and FALSE otherwise. Its primary purpose is to display information about the program or the development team in response to user interaction with the About box.

DrawRectangle Function:

The DrawRectangle function is responsible for rendering a rectangle on the window, representing a beat in the drum sequence.

It receives parameters such as the device context (HDC), coordinates (x and y), and bitmasks (dwSeqPerc and dwSeqPian) indicating the status of percussion instruments for a given beat.

The function determines the color of the rectangle based on the instrument status and uses Rectangle to draw it.

This function plays a crucial role in visually representing the state of the drum sequence within the window.

ErrorMessage Function:

The ErrorMessage function displays an error message box to the user. It takes parameters such as the window handle (hwnd), an error message (szError), and the current file name (szTitleName).

The error message is constructed using wsprintf, including the current file name if available, and then presented to the user via MessageBox. This function also utilizes MessageBeep to provide an audible cue for the error.

DoCaption Function:

The DoCaption function sets the window caption based on the current file name. It utilizes wsprintf to construct the caption string, incorporating the file name or indicating that the file is untitled.

The resulting string is then set as the window caption using SetWindowText. This function ensures that the window title accurately reflects the state of the loaded or saved file.

AskAboutSave Function:

The AskAboutSave function prompts the user with a message box, asking whether to save changes before proceeding with certain actions.

It constructs the message using wsprintf, including the current file name, and utilizes MessageBox to interact with the user.

The function returns the user's choice (yes, no, or cancel), influencing subsequent program behavior based on the response.

These functions collectively contribute to the core functionality and user interaction of the drum machine program, providing a well-structured and responsive user interface.

In summary, the "DRUM" program exhibits a well-organized structure and functionality. The program flow can be outlined as follows:

Initialization:

- The program initializes the parameters of the drum machine, including tempo, velocity, and the number of beats.
- GUI elements, such as scroll bars and menu items, are set up to provide user interaction.

User Interactions:

- The program handles various user interactions through the window procedure (WndProc), responding to menu commands, mouse events, and scroll bar adjustments.
- Menu commands allow users to perform actions such as creating a new file, opening/saving files, starting/stopping sequences, and displaying an About box.
- Mouse interactions, particularly left and right button clicks, modify the drum sequences, enabling users to create and edit rhythmic patterns.
- Scroll bar adjustments control parameters like tempo and velocity, providing real-time adjustments to the playback characteristics.

Sequence Modification:

- The drum sequences are modified based on user interactions, with each beat in the sequence representing a percussion instrument.
- The DrawRectangle function visually represents the state of the drum sequence, ensuring a clear and intuitive display of the rhythm.

Custom Messages:

- Custom Windows messages (WM_USER_NOTIFY, WM_USER_FINISHED, WM_USER_ERROR) are employed for specific functionalities, possibly related to MIDI sequence notifications and error handling.

File I/O Operations:

- Custom functions for file I/O operations (DrumFileOpenDlg, DrumFileRead, DrumFileWrite) are used to open, read, and write drum machine configurations. These functions contribute to the persistence of user-created drum sequences.

Sequence Control:

- Functions like DrumBeginSequence and DrumEndSequence manage the initiation and termination of MIDI sequences. These functions likely play a role in controlling the playback of drum sequences.

Error Handling:

- The ErrorMessage function provides a mechanism for displaying informative error messages to the user, enhancing the program's usability by offering insights into potential issues.

About Box:

- The AboutProc function serves as the dialog procedure for the About box, allowing users to access information about the program or its developers.

This breakdown highlights the program's modular design, separating different aspects of functionality into distinct sections. The effective use of custom messages, functions, and clear user interactions contributes to a robust and user-friendly drum machine application.

DRUMFILE.C PROGRAM

The provided portion of code is from the "DRUMFILE.C" file, responsible for handling file input/output operations in the DRUM program. Let's break down the code into paragraphs, focusing on its structure and purpose.

Header Inclusions:

The code begins with the inclusion of necessary headers, namely <windows.h>, <commdlg.h>, and the custom headers for the drum program, "drumtime.h" and "drumfile.h." These headers provide essential functionalities for Windows programming and definitions specific to the DRUM program.

Global Variables and Definitions:

Following the header inclusions, the code declares and initializes global variables and definitions that are crucial for file handling. These include the OPENFILENAME structure (ofn), an array defining file filters (szFilter), and several strings identifying specific components in the DRUM file format.

File Handling Routines:

The code then defines routines related to file input/output operations. It initializes the OPENFILENAME structure (ofn) with appropriate values, including the file filter, default extension, and flags. The function DrumFileOpenDlg is designed to open a file dialog for selecting or creating DRUM files. It takes parameters such as the parent window handle, file name, and title name.

Function Details:

- ofn.hwndOwner = hwnd;: Sets the owner window for the file dialog.
- ofn.lpstrFilter = szFilter[0];: Specifies the file filter for DRUM files.
- ofn.lpstrFile = szFileName;: Sets the buffer to store the selected file name.
- ofn.nMaxFile = MAX_PATH;: Defines the maximum length of the file name.
- ofn.lpstrFileTitle = szTitleName;: Sets the buffer to store the file title.
- ofn.nMaxFileTitle = MAX_PATH;: Defines the maximum length of the file title.
- ofn.Flags = OFN_CREATEPROMPT;: Configures the file dialog to prompt for file creation if the selected file does not exist.
- ofn.lpstrDefExt = TEXT("drm");: Sets the default extension for DRUM files.

Return Statement:

The function returns the result of the GetOpenFileName function, indicating whether the user successfully selected or created a DRUM file.

This portion of the code establishes the framework for opening DRUM files through a file dialog, ensuring that the selected file adheres to the DRUM file format. It encapsulates the necessary configurations and behaviors for interacting with the file dialog in a Windows environment.

File Save Dialog Function (DrumFileSaveDlg):

The function initiates a file save dialog to allow the user to specify the name and location for saving a DRUM file.

It utilizes the OPENFILENAME structure (ofn) initialized with relevant parameters such as the owner window, file filter, default extension, and flags. The function returns a Boolean value indicating the success of the operation.

File Write Function (DrumFileWrite):

This function is responsible for writing the DRUM structure and associated information to a file. It takes as parameters the DRUM structure (pdrum) and the file name (szFileName).

The function begins by creating and opening a new file for writing using the Windows Multimedia I/O API (mmioOpen). It then proceeds to structure the file content with specific chunks and sub-chunks required for the DRUM file format.

File Structure Creation:

- ✓ The function starts by creating a "RIFF" chunk with a "CPDR" type, representing the top-level structure of a DRUM file.
- ✓ Within the "RIFF" chunk, it creates a "LIST" sub-chunk with an "INFO" type, encapsulating additional information about the file.
- ✓ Within the "INFO" chunk, it creates a sub-sub-chunk with an "ISFT" type, containing information about the software used to create the file.
- ✓ It writes the software information to the file and ascends from the "ISFT" sub-sub-chunk.
- ✓ The function also creates a sub-sub-chunk with an "ISCD" type, containing the current date.
- ✓ It writes the date information to the file and ascends from the "ISCD" sub-sub-chunk.
- ✓ The function then ascends from the "INFO" chunk.
- ✓ Next, it creates a "fmt " sub-chunk, representing the format of the data to follow, and writes the format information to the file.
- ✓ Following that, it creates a "data" sub-chunk and writes the DRUM structure to the file.

Error Handling and Cleanup:

The function handles errors during the file writing process, deleting the file if an error occurs. It uses specific error messages (szErrorNoCreate and szErrorCannotWrite) to communicate issues related to file creation and writing.

In summary, this code segment encapsulates the logic for saving DRUM data to a file. It ensures proper structuring of the file content according to the DRUM file format and includes error handling to manage potential issues during the file writing process.

Function Overview (DrumFileRead):

The function is designed to read DRUM data from a file specified by szFileName. It takes as parameters a pointer to the DRUM structure (pdrum) where the data will be stored.

File Reading Process:

The function starts by declaring local variables, including the DRUM structure (drum), an integer for the file format (iFormat), and an array of MMCKINFO structures (mmckinfo).

It initializes the MMCKINFO array and opens the specified file for reading using the Windows Multimedia I/O API (mmioOpen).

The function then searches for a "RIFF" chunk with a "DRUM" form-type, using mmioDescend and mmioStringToFOURCC. If not found, it returns an error indicating that the file is not a standard DRUM file.

Reading "fmt " Sub-chunk:

The function locates and descends into the "fmt " sub-chunk, verifying its size and reading the file format information. If the format is unsupported or cannot be read, it returns an appropriate error message.

It checks if the format is either 1 or 2, and if not, it returns an error indicating an unsupported format.

Reading "data" Sub-chunk:

The function then locates and descends into the "data" sub-chunk, verifying its size and reading the DRUM structure. If the size is incorrect or reading fails, it returns an error.

Conversion for Format 1:

If the file format is 1, the function performs a conversion by rearranging sequence data in the DRUM structure.

Closing the File and Returning:

Finally, the function closes the file using mmioClose and copies the read DRUM data to the provided pointer (pdrum).

Error Handling:

Throughout the process, the function incorporates error handling, closing the file and returning specific error messages in case of issues.

Let's summarize the key points:

Interface Overview:

- The program interface initially displays 47 percussion instruments on the left side, arranged in two columns.
- The right side contains a grid representing percussion sounds over time, with each instrument associated with a row and each column representing a beat.

Sequence Playback:

- Selecting "Running" from the Sequence menu attempts to open the MIDI Mapper device.
- A "bouncing ball" moves across the grid, playing percussion sounds for each beat.
- Left-clicking adds dark gray squares for percussion sounds, right-clicking adds light gray squares for piano beats, and both buttons together create black squares with both sounds.

Repeat Sign and Beats:

- Dots above the grid mark every 4 beats for easy reference.
- A repeat sign (:|) at the upper right corner indicates the sequence's length.
- Clicking above the grid places the repeat sign, and the sequence plays up to, but not including, the beat under the repeat sign.

Control Elements:

- Horizontal scroll bar controls velocity (affecting volume and, in some synthesizers, timbre).
- Vertical scroll bar controls tempo on a logarithmic scale, ranging from 1 second per beat at the bottom to 10 milliseconds per beat at the top.

File Management:

- The File menu allows saving and retrieving files with a .DRM extension, using the RIFF file format.

About and Stopped Options:

- The About option in the Help menu provides a brief summary of mouse usage and scroll bar functions.
- The Stopped option in the Sequence menu stops the music and closes the MIDI Mapper device after finishing the current sequence.

Multimedia Time Functions:

- In DRUM.C, the absence of calls to multimedia functions is highlighted, with the real-time functionality being delegated to the DRUMTIME module.
- The Windows timer, though simple, is deemed inadequate for time-critical applications like playing music.
- To address this, the multimedia API provides a high-resolution timer implemented through functions with the prefix "time."
- The multimedia timer works with a callback function running in a separate thread. Two critical parameters are specified: delay time and resolution. The resolution acts as a tolerable error, allowing flexibility in the actual timer delay.
- Before using the timer, obtaining device capabilities through timeGetDevCaps is recommended. This provides minimum and maximum resolution values, which can be used in subsequent timer function calls.
- The sequence involves calling timeBeginPeriod to set the required timer resolution, followed by setting a timer event using timeSetEvent. To stop events, timeKillEvent is employed. The multimedia timer is designed specifically for playing MIDI sequences and offers limited functionality for other purposes.

DRUMTIME Module:

- The DRUMTIME module is crucial for handling time-related operations in the DRUM program. It manages the multimedia timer, orchestrating events based on user interactions and MIDI sequence requirements.
- The DrumSetParams function in DRUM.C calls the equivalent function in DRUMTIME.C, passing a pointer to a DRUM structure.

- This structure contains essential parameters such as beat time, velocity, number of beats, and sequences for percussion and piano sounds. This information is crucial for setting up and modifying the drum sequences.
- DrumBeginSequence is invoked to initiate the MIDI sequence. It opens the MIDI Mapper output device, sets up the timer with specified resolution and delay, and selects instrument voices for percussion and piano.
- The DrumTimerFunc callback is at the heart of the timer functionality. It handles MIDI Note On/Off messages, updates the beat index, and triggers the next timer event.
- To conclude the sequence, DrumEndSequence is called, either ending it immediately (TRUE argument) or allowing it to complete the current cycle before termination (FALSE argument).
- This function manages the cleanup process, including killing the timer event, ending the timer period, sending "all notes off" messages, and closing the MIDI output port.

In summary, the Multimedia Time Functions and the DRUMTIME module work synergistically to provide accurate timing for playing MIDI sequences in the DRUM program. The modular design enhances code organization and readability, making the program efficient and responsive to user interactions.

The section you provided explains the RIFF (Resource Interchange File Format) file I/O in the DRUM program, detailing how the program saves and retrieves files containing information stored in the DRUM structure. The RIFF format is a tagged file format, where data is organized in chunks, each identified by a 4-character ASCII tag.

RIFF File Structure:

- Chunks: The RIFF file consists solely of chunks, each composed of a chunk type (4-character ASCII tag), chunk size (32-bit value indicating the size of the chunk data), and the actual chunk data. Chunks are word-aligned, and the chunk size does not include the 8 bytes required for the chunk type and size.
- RIFF and LIST Chunks: There are two special types of chunks - RIFF chunks and LIST chunks. A RIFF chunk is used for the overall file, and a LIST chunk is used to consolidate related sub-chunks within the file.

mmio Functions:

- The multimedia API includes 16 functions beginning with the prefix mmio, specifically designed for working with RIFF files.
- To open a file, mmioOpen is used. mmioCreateChunk creates a chunk, mmioWrite writes chunk data, and mmioAscend finalizes the chunk.

- Nested levels of chunks are maintained with multiple MMCKINFO structures associated with each level. In the DRUM program, three levels of chunks are used.
- The RIFF file begins with a RIFF chunk, starting with the string "RIFF" and a 32-bit value indicating the size of the file.

Writing RIFF File (DrumFileWrite):

- The function begins by creating the RIFF chunk using mmioCreateChunk with mmckinfo[0]. Subsequently, a LIST chunk is created within the RIFF chunk using mmckinfo[1], and an ISFT sub-chunk for software identification is created with mmckinfo[2].
- Further chunks like ISCD, "fmt," and "data" are created within the LIST chunk. mmioAscend is used to finalize each chunk and move to the next level.
- Finally, the size of the overall RIFF chunk is filled in with one last call to mmioAscend using mmckinfo[0]. mmioClose concludes the file-writing process.

Reading RIFF File (DrumFileRead):

- Reading involves using mmioRead instead of mmioWrite and mmioDescend instead of mmioCreateChunk. The process is similar to writing, but it descends into chunks to read data.
- The program can identify the format identifier and convert files written with an earlier format to the current one.

In summary, the DRUM program utilizes the RIFF file format and mmio functions for efficient and flexible file I/O operations. The use of chunks allows easy extensibility of the file format, and the mmio functions streamline the process of reading and writing these structured files.

That's it for the second last chapter in WinAPI GUI Programming, let's finish this chapter 23! Let's go!!👍👍😅