DEVELOPER HISTORY BRIEF · 1992–2026

The Windows API & the Developer Toolchain

From 16-bit message loops on Windows 3.1 to the WinUI 3 stack of Windows 11 — how the Windows API evolved across seven generations, and which languages, frameworks and tools defined each era of Windows development.

WIN16 | WIN32 | COM / ACTIVEX | .NET | WPF / XAML | WINRT / UWP | WINUI 3

7 API GENERATIONS

9 DEVELOPMENT ERAS

34 YEARS OF EVOLUTION

Compiled **October 2026** · covering Windows 3.1 through Windows 11 · each era documents the state of the API, the preferred languages, the dominant tools, and the force that drove the next shift.

THE WINDOWS API · 1992–2026

Seven Generations of the Windows API

Dominant API paradigm per era with its preferred languages and tools · cyan dots mark Windows release milestones on the 34-year spine below.

1992–1995
Win16

API
16-bit C API in KERNEL, USER & GDI — message loop, WndProc, far pointers, cooperative multitasking

LANGUAGES
C · Pascal · Basic

TOOLS
MS C/C++ 7.0, VC++ 1.x, VB 1–3, Borland C++, PowerBuilder

1993–1996
Win32

API
32-bit, Unicode, preemptive multitasking; Win32s subset shim for 3.1

LANGUAGES
C · C++ · Object Pascal

TOOLS
VC++ 2.0/4.x + MFC 4.x, ATL 1.0, VB 4, Delphi 1–2

1996–2000
COM / ActiveX

API
COM & OLE 2 components, IDispatch Automation, ActiveX controls; COM+ (2000)

LANGUAGES
C++ · Basic · VBScript

TOOLS
VC6 + MFC 6 / ATL 3, VB 5–6, VS 6.0, Delphi 3–5

2001–2005
.NET / WinForms

API
.NET Framework CLR, Windows Forms, GDI+ — managed code layered over Win32

LANGUAGES
C# · VB.NET · C++/MFC

TOOLS
VS .NET 2002–2005, Platform SDK, Delphi 7

2006–2011
WPF / XAML

API
.NET 3.0: WPF, WCF, WF — XAML markup, DirectX-composited UI; Ribbon (Win7)

LANGUAGES
C# 3–4 · VB 9 · C++/CLI

TOOLS
VS 2008–2010, Expression Blend, Qt

2012–2020
WinRT / UWP

API
WinRT: object-oriented, metadata-based, async-first; UWP packages, MSIX

LANGUAGES
C# · C++/CX → C++/WinRT · JS

TOOLS
VS 2012–2019, .NET Core 3.0, Electron + WebView2

2021–2026
WinUI / App SDK

API
WinUI 3 + Windows App SDK — UI stack decoupled from the OS; WebView2

LANGUAGES
C# (.NET 6–8) · C++/WinRT · Rust

TOOLS
VS 2022, VS Code, .NET MAUI, Tauri



WINDOWS RELEASE MILESTONES · PROPORTIONAL TIME AXIS, 1992 → 2026

THE PARADIGM HAND-OFF

Win16 → Win32 → COM / ActiveX → .NET / WinForms → WPF / XAML → WinRT / UWP → WinUI 3 / App SDK

● Windows release ▢ API generation + toolchain ⬭ Paradigm hand-off

Win32 is never retired — every generation layers on top of it.

OVERVIEW

One Platform, Seven API Generations

This document traces the Windows API from **Windows 3.1 (April 1992)** to the **Windows 11 era of 2026**: seven API generations, nine development eras, the languages and tools that dominated each of them, and the forces that pushed each transition. The full-page timeline chart that precedes this chapter is the visual map; the chapters that follow expand each era in detail, and a reference table near the end collects the key dates.

The organizing insight is that Windows development is **cumulative rather than replacement-driven**. No API generation was ever fully retired: Win32 remains the substrate beneath .NET, WinRT and WinUI, and a 1995-era Win32 program still runs unmodified on Windows 11. Each era therefore layers on top of the last instead of erasing it — backwards compatibility being Windows' strategic moat. That layering explains both the relentless growth of the API surface and the consolidation of tooling around Visual Studio. Every era chapter below follows the same pattern: the state of the API, the languages developers preferred, the tools they actually used, and a short why it changed note that leads into the next era.

ERA 01 · 1992–1995

Windows 3.1 and the Win16 Era

Windows 3.1 shipped in April 1992 onto a 16-bit, cooperatively multitasking operating system layered over DOS. Its API — retroactively named **Win16** — was a flat C API living in three core libraries: KERNEL (memory, tasks, files), USER (windows, messages, menus, controls) and GDI (graphics, fonts, printing). Programming meant registering a window class, writing a window procedure (**WndProc**) and pumping a message loop; everything else was built from handles such as **HWND** and **HDC**, near/far pointers against segmented 64 KB memory, and the era's beloved-or-cursed Hungarian notation. It was low-level, demanding work by any standard.

C was the dominant language, with Pascal a respectable second, and the Windows SDK was still a separate product from the compiler. Professionals used **Microsoft C 7.0** and the 16-bit **Visual C++ 1.x** releases; **Borland C++ 3.1/4.0** was the cherished alternative and **Turbo Pascal for Windows** served the Pascal camp. The mass-market shock came from **Visual Basic** — born in 1991 out of Alan Cooper's "Ruby" prototype and matured by **VB 3.0 (1993)**: painting forms on a canvas and wiring up BASIC code made Windows GUI programming accessible to an order of magnitude more developers. **PowerBuilder** and dBASE-family tools owned much of the database front-end niche.

API	Win16 — KERNEL / USER / GDI, message loop + WndProc, handles, far pointers, cooperative multitasking
LANGUAGES	C (dominant) · Pascal · Basic
TOOLS	Microsoft C/C++ 7.0 · Visual C++ 1.x (16-bit) + Windows SDK · Borland C++ 3.1/4.0 · Visual Basic 1–3 · PowerBuilder

WHY IT CHANGED

Win16 had hard architectural ceilings: segmented 16-bit memory, cooperative multitasking where one misbehaving application froze the machine, and DOS heritage everywhere. The NT team set out to build a portable, secure, 32-bit operating system — and the 386/486 generation finally made it affordable. The programming model would carry forward; the runtime foundations would be replaced.

ERA 02 · 1993–1996

Windows NT and the Birth of Win32

Windows NT 3.1 (July 1993) introduced **Win32**: a clean 32-bit API with flat memory, preemptive multitasking, **Unicode from day one**, and true portability — it shipped for MIPS, Alpha and later PowerPC alongside x86. Crucially, Microsoft kept the programming model identical: messages, handles and window procedures, so Win16 source code ported with largely mechanical changes. For developers still on 3.1, the **Win32s** subset (1994) provided a shim, letting a single codebase target both worlds.

Visual C++ 2.0 (1994) was the first 32-bit-only Visual C++, targeting NT exclusively, and the professional toolchain consolidated around VC++ plus **MFC** — a class framework that wrapped the raw API into C++ objects — plus the SDK for the undiluted view.

API	Win32 — 32-bit flat memory, Unicode, preemptive multitasking, hardware portability; Win32s subset shim for Windows 3.1
LANGUAGES	C · C++ (with class frameworks)
TOOLS	Visual C++ 2.0 (first 32-bit) · MFC · Windows SDK · Borland C++ / Watcom C++ following to 32-bit

WHY IT CHANGED

486 and Pentium hardware finally made 32-bit computing affordable, corporate buyers demanded NT's reliability and protection model, international markets pushed Unicode, and a portable kernel hedged against any single CPU vendor. Win32 became the platform's long-term contract — but consumers were still running 16-bit Windows until 1995.

ERA 03 · 1995–1996

Windows 95: Win32 Goes Mainstream

Windows 95 (August 1995) took Win32 to the consumer mass market and enriched it: common controls (comctl32 — toolbars, status bars, tree views), common dialogs, the **Registry**, long filenames and **Winsock 1.1** for TCP/IP. That same September, **DirectX** debuted as the "Game SDK" — DirectDraw and DirectSound — in a deliberate campaign to win game developers back from DOS. In 1996, **Windows NT 4.0** ported the Win95 interface and DirectX onto the NT kernel, beginning the long convergence of the two product lines.

The languages of the day were C, C++ with class frameworks, Object Pascal and Basic. **Visual C++ 4.x with MFC 4.x** was the professional C++ pairing, and **ATL 1.0 (1996)** appeared for lean component work. **Visual Basic 4.0** delivered the first 32-bit VB with class modules. The era's sensation was **Delphi 1 (February 1995)** — Object Pascal with a form designer and a lightning-fast native compiler — followed by the fully 32-bit **Delphi 2 (1996)**. Borland C++ 5.0 and Watcom C++ (a favorite in games) rounded out the field.

API	Win32 mainstream — comctl32 common controls, Registry, Winsock, DirectX 1.0 (1995); Win95 UI + DirectX ported to NT 4.0 (1996)
LANGUAGES	C · C++ with frameworks · Object Pascal · Basic
TOOLS	Visual C++ 4.x + MFC 4.x · ATL 1.0 · Visual Basic 4.0 · Delphi 1–2 · Borland C++ 5.0 · Watcom C++

WHY IT CHANGED

Consumers needed 32-bit power without NT's hardware appetite, the internet was arriving on desktops through Winsock, and multimedia gaming demanded DirectX. Meanwhile component software was brewing in the background — COM was about to become the story.

ERA 04 · 1996–2000

The COM / ActiveX Component Era

While the OS matured through **Windows 98** (WDM driver model, USB) and **Windows 2000** (February 2000), the API story was **COM**: a binary component standard that let C++ classes be consumed from Visual Basic, Delphi and scripting languages alike. OLE 2 compound documents, OLE Automation (the IDispatch dispatch interface) and **ActiveX controls** (the 1996 rebranding of OLE controls) meant components you could drop onto a VB form or embed in a web page. **COM+** arrived with Windows 2000, layering MTS-style transactional services onto the model.

On the tooling side this was the golden age of the two-IDE duopoly. **Visual C++ 5.0 and 6.0** — VC6 shipping with MFC 6.0 and ATL 3.0 — became the longest-lived C++ tool in mainstream history. **VB5 (1997)** added native code compilation and **VB6 (1998)** made millions of business developers productive. **Visual Studio 97** unified the suite for the first time and **VS 6.0 (1998)** defined an era; **Delphi 3–5** thrived with interfaces and COM support; **VBA** carried automation into every corner of Office. VBScript and JScript powered ASP web pages and Windows Script Host.

API	COM / OLE 2 / ActiveX — binary components, IDispatch Automation, COM+ services (2000); Win98 adds WDM + USB
LANGUAGES	C++ for components · Basic for the masses · VBScript / JScript for ASP and WSH
TOOLS	Visual C++ 5/6 (MFC 6, ATL 3) · Visual Basic 5–6 · Visual Studio 97 / 6.0 · Delphi 3–5 · VBA

WHY IT CHANGED

"DLL hell" and the need for binary reuse across languages and teams demanded a stable ABI contract — COM was that contract. Enterprise multi-tier computing and the web wave pushed ActiveX into pages (later largely abandoned for security reasons, though COM itself persisted everywhere and still underpins the Windows Shell today). The next shift would be bigger: a managed runtime.

ERA 05 · 2001–2005

Windows XP and the .NET Turn

Windows XP (October 2001) merged the consumer and NT lines at last: stable, themed through comctl32 v6 visual styles, with **GDI+** for modern 2D graphics. With the core Win32 API mature and largely frozen, new platform investment moved up a level: the **.NET Framework 1.0 (February 2002)** — a managed runtime (CLR) with garbage collection, assemblies and metadata — and **Windows Forms** as its UI layer over Win32. **C#**, a new language led by Anders Hejlsberg (of Delphi fame), became the strategic language; **VB.NET** modernized Visual Basic at the cost of breaking VB6 compatibility — one of

the most controversial moves in developer-tools history, and one whose aftermath (VB6 code living on for decades) proved how deep tool loyalty runs.

Visual Studio .NET 2002/2003 and **VS 2005** (.NET 2.0, generics) carried the managed wave, while C++ with MFC/ATL remained the native path for systems work and the Platform SDK served Win32 holdouts — VC6, famously, refused to die in the field. **Delphi 7 (2002)** was the last great classic Delphi as Borland wound down its developer tools, selling them in 2006 into what became CodeGear and then Embarcadero.

API	.NET Framework CLR — Windows Forms, GDI+, ASP.NET; managed code layered over a stable Win32 (visual styles, GDI+)
LANGUAGES	C# (new, strategic) · VB.NET (breaking from VB6) · C++ with MFC/ATL for native
TOOLS	VS .NET 2002–2003 · VS 2005 (.NET 2.0, generics) · Platform SDK · VC6 in the field · Delphi 7

WHY IT CHANGED

Managed code promised memory safety, assembly versioning (an answer to DLL hell) and a productivity leap; Java's rise demanded a response; the web was becoming the default delivery model (ASP.NET); and Microsoft needed one runtime to unify the VB and C++ worlds. The CLR era had begun — the next act would be about how its UIs looked.

ERA 06 · 2006–2011

Vista and Windows 7: The WPF / XAML Wave

Windows Vista (business November 2006, consumer January 2007) shipped with **.NET Framework 3.0**, whose crown jewel was **WPF**: user interfaces declared in XAML markup and rendered through DirectX compositing — resolution independence, vector drawing, hardware acceleration. WCF (communication) and WF (workflow) rode along. Win32 remained the core OS API — Vista added UAC integrity levels and session 0 isolation for services — and **Windows 7 (2009)** refined the stack and added the **Windows Ribbon framework**.

C# 3.0 (2007, LINQ) led the languages; **C++/CLI** bridged managed and native; MFC/ATL were refreshed — VS 2008 SP1 even granted MFC an Office-style ribbon. **VS 2008** shipped first-class WPF/XAML editors and **VS 2010** was itself rewritten in WPF; **Expression Blend** gave designers a parallel XAML tool. WinForms stayed the enterprise workhorse throughout, **Qt** and wxWidgets served portable C++, and the first web-engine embedding (**CEF**, 2008) quietly foreshadowed Electron.

API	.NET 3.0 — WPF (XAML, DirectX-composited UI), WCF, WF; Win32 core + UAC; Ribbon framework (Win7, 2009)
LANGUAGES	C# 3–4 (LINQ) · VB 9 · C++/CLI · refreshed MFC/ATL
TOOLS	VS 2008–2010 · Expression Blend · WinForms (workhorse) · Qt / wxWidgets · CEF embedding

WHY IT CHANGED

GPUs made DirectX-composited UIs natural, HD displays demanded resolution independence, and the designer/developer split called for a declarative UI format designers could own. WPF won hearts but adoption was gradual — WinForms codebases were enormous and .NET 3.x still ran on the same CLR — so both stacks coexisted through the whole period.

ERA 07 · 2012–2014

Windows 8 and WinRT

Windows 8 (October 2012) was the biggest platform break since 1995: a touch-first, full-screen "Store" app model built on **WinRT (Windows Runtime)** — a new object-oriented, metadata-described, async-first API layered over Win32. Because the API surface carried full type metadata, Microsoft could project it natively into four sanctioned languages: C#, VB.NET, C++/CX, and JavaScript + HTML5. Sandbox privileges and Windows Store distribution changed how apps were installed and trusted, while classic desktop applications remained fully supported on the side.

Tooling was **Visual Studio 2012/2013** with Blend again prominent for XAML design. Adoption was rocky: developers with large desktop investments balked at Metro-only constraints and the desktop/tablet split, and Windows 8.1 (2013) walked several constraints back. The strategic lesson — that developers needed a bridge rather than a wall — shaped everything Windows 10 would do.

API	WinRT — object-oriented, metadata-based, async-first; XAML or HTML/JS front ends; sandboxed Store distribution; Win32 still fully supported
LANGUAGES	C# · VB.NET · C++/CX · JavaScript + HTML5 (the four sanctioned Metro languages)
TOOLS	Visual Studio 2012–2013 · Blend for XAML

WHY IT CHANGED

The iPad shock and mobile-first computing forced a touch-first redesign; curated stores plus sandboxing answered security and distribution problems; and a metadata-driven API meant no single language would be second-class. But the transition overestimated how fast desktop developers would move — and Windows 10 corrected course.

ERA 08 · 2015–2020

Windows 10 and UWP

Windows 10 (July 2015) introduced **UWP (Universal Windows Platform)**: one packaged application model across desktops, tablets, Xbox, HoloLens and IoT, with the XAML UI stack (the WinUI library, later spun out of the OS), **MSIX** packaging, and the **Desktop Bridge (2016)** letting existing Win32 applications enter the Store. Crucially, classic Win32 desktop development remained first-class — Windows 10 was a re-platforming offer, not a wall.

C# was the UWP default; on the native side, **C++/WinRT** (public from 2017) — a header-only, standard C++17 projection — replaced C++/CX as Microsoft's recommended C++ path. Tooling ran through **VS 2015/2017/2019**. Off the UWP track, the current era's biggest current was **.NET Core (2016)**: cross-platform, open-source .NET — and **.NET Core 3.0 (September 2019)** brought WinForms and WPF back onto modern .NET. Meanwhile **Electron** — with **WebView2**, Edge/Chromium hosting, arriving from 2020 — made web technology the pragmatic default for many new Windows desktop apps, and React Native and Flutter reached for Windows as well.

API	UWP — one app model across device families; WinUI XAML, MSIX packaging, Desktop Bridge (2016); Win32 desktop remains first-class
LANGUAGES	C# (UWP default) · C++/WinRT (standard C++17, from 2017) · JS/TS via web stacks
TOOLS	VS 2015–2019 · .NET Core 2016 → 3.0 (WinForms/WPF return) · Electron + WebView2 · React Native / Flutter

WHY IT CHANGED

One binary across device families was the mobile-era dream, but UWP's restrictions never fully won over desktop developers, and OS-locked frameworks iterated too slowly. Open-sourcing .NET and decoupling the UI stack from Windows releases were the answers to developer flight toward cross-platform stacks.

ERA 09 · 2021–2026

Windows 11: WinUI 3 and the Windows App SDK

Windows 11 (October 2021) arrived alongside the **Windows App SDK** (Project "Reunion", GA 2021–22), which delivered **WinUI 3**: the modern XAML stack finally decoupled from OS release cycles. The eternal substrate is unchanged — **Win32/COM power everything** — while **WebView2** became the standard embedded-web surface and MSIX the default packaging story.

The language field is the widest it has ever been: **C# on unified .NET (5 → 6/7/8)** leads most new development; **C++/WinRT** anchors native work; Microsoft officially blessed **Rust** through the windows-rs projections (2021); web stacks (Electron, Tauri, React Native) carry much cross-platform traffic; and **.NET MAUI (2022)** extends Xamarin's cross-platform role. Tooling centers on **VS 2022** (the first 64-bit release) and VS Code, with WPF/WinForms on .NET 8 still the enterprise default, WinUI 3 the pick for new Windows-native UI — and AI assistance (the Copilot era) reshaping daily workflow.

API	WinUI 3 + Windows App SDK — UI stack decoupled from the OS; Win32/COM backbone; WebView2; MSIX everywhere; windows-rs (Rust) official
LANGUAGES	C# (.NET 6–8) · C++/WinRT · Rust (windows-rs) · web stacks (Electron / Tauri / React Native)
TOOLS	VS 2022 (64-bit) · VS Code · .NET MAUI · WPF/WinForms on .NET 8 (enterprise default) · WinUI 3

WHY IT CHANGED

Frameworks must iterate faster than the OS ships; packaging needed to become universal; memory-safe languages answered mounting security pressure; and pragmatic hybrid apps (web UI in a native shell) won much of the new-build market. Through every one of these shifts, the Win32 contract held — making it arguably the longest-lived continuously supported application API in mainstream computing.

REFERENCE · 1992–2026

Key Dates at a Glance

The table compresses the timeline into its load-bearing milestones: the Windows release or platform event, the API development it marked, and the tool most developers would have reached for at the time.

YEAR	WINDOWS / PLATFORM	API MILESTONE	FLAGSHIP TOOLS
1992	Windows 3.1	Win16 API in KERNEL / USER / GDI	VC++ 1.x, VB 1–3, Borland C++ 3.1
1993	Windows NT 3.1	Win32 debuts: 32-bit, Unicode	VC++ 1.0, MS C/C++ 7.0
1994	Win32s for 3.1	Win32 subset on 16-bit Windows	VC++ 2.0 (first 32-bit only)
1995	Windows 95	Win32 mainstream; DirectX 1.0; comctl32	Delphi 1, VB 4.0, VC++ 4.0
1996	Windows NT 4.0	Win95 UI + DirectX on NT; ATL 1.0	VC++ 4.x, MFC 4.x
1997	ActiveX wave	OLE Automation components in pages	VS 97, VB 5.0 (native code)
1998	Windows 98	WDM driver model, USB	VS 6.0 — VC6, VB6

YEAR	WINDOWS / PLATFORM	API MILESTONE	FLAGSHIP TOOLS
2000	Windows 2000	COM+ services	VC6 dominant in the field
2001	Windows XP	GDI+, visual styles (comctl32 v6)	VS 6.0 still dominant
2002	.NET Framework 1.0	WinForms; C# 1.0; VB.NET	VS .NET 2002
2005	.NET Framework 2.0	Generics, 64-bit CLR	VS 2005
2006–07	Windows Vista	.NET 3.0: WPF, WCF, XAML	VS 2008 (Nov 2007)
2009	Windows 7	Windows Ribbon framework	VS 2010 (C# 4.0)
2012	Windows 8	WinRT; async-first; Store apps	VS 2012
2015	Windows 10	UWP app model; Desktop Bridge (2016)	VS 2015
2017	.NET Core 2.0 era	C++/WinRT replaces C++/CX	VS 2017
2019	.NET Core 3.0	WinForms / WPF go open source	VS 2019
2021	Windows 11	windows-rs (Rust); WinUI 3 previews	VS 2022 (64-bit)
2022	Windows App SDK 1.x	WinUI 3 stable; .NET MAUI ships	VS 2022
2024–26	.NET 8 LTS era	WebView2 standard in new builds	VS 2022, VS Code

SYNTHESIS

The Big Picture

Read as one arc, the Windows API is a story of **successful layers**. The paradigm chain below is the whole history in one line: each stage wrapped the previous one rather than replacing it — the chain is additive, and Win32 at its center has never been retired.

API PARADIGM CHAIN · 1992 → 2026

Win16 → Win32 → COM / ActiveX → .NET / WinForms → WPF / XAML → WinRT / UWP → WinUI 3 / App SDK

The language torch passed through three great hands: from **C** in the segmented-memory days, to the **C++ / Visual Basic / Delphi** triad of the framework and component decades, to **C#**'s long managed reign — now flanked by **C++/WinRT** and **Rust** on the native side and by web technology everywhere pragmatism wins. The tool story is one of consolidation: a scattered SDK-plus-compiler landscape in 1992 converged, within six years, onto Visual Studio — which has held the center ever since.

LANGUAGE TORCH

C → C++ · VB · Delphi → C# · VB.NET → C# · C++/WinRT · Rust · Web

Three patterns are worth carrying away. First, **compatibility is the strategy**: every generation layered onto its predecessor, and the 1993 Win32 contract still governs Windows 11. Second, **each wave mirrors the computing mainstream** of its moment — 16-to-32-bit PCs, binary components, managed code, touch and app stores, and finally decoupled cross-platform frameworks. Third, for anyone choosing a stack today it doubles as a decision guide: **C# on .NET 8 with WPF, WinForms or WinUI 3** for Windows-native work, **C++/WinRT** for systems code, web technology for cross-platform reach — with Rust a credible fourth way on the native edge.