Contents

- API index
  - Windows API list
  - Windows API sets
    - Overview
    - API set loader operation
    - Detect API set availability
    - Windows umbrella libraries
  - WinRT APIs callable from a desktop app

API Index for desktop Windows applications

11/2/2020 • 2 minutes to read • Edit Online

This article provides links to reference documentation for APIs that can be used in desktop Windows apps.

Win32 (Windows API)

The Win32 API (also called the Windows API) is the native platform for Windows apps. This API is best for desktop apps that require direct access to system features and hardware. The Windows API can be used in all desktop apps, and the same functions are generally supported on 32-bit and 64-bit Windows.

- Win32 API reference by feature
- Win32 API reference by header
- Win32 and COM APIs for UWP apps
- Windows umbrella libraries
- Windows API Sets

Windows Runtime (WinRT)

WinRT is the leading edge platform for Windows 10 apps and games, including desktop apps. The WinRT API is suitable for both native C++ and managed desktop apps that require a sophisticated UI, styles customization, and graphics-intensive scenarios.

- WinRT API reference
- WinRT APIs callable from Win32, WPF, and Windows Forms desktop apps

.NET

The .NET class libraries provide access to Windows system and UI features for managed desktop apps, including WPF and Windows Forms apps.

- .NET API

Windows API index

1/14/2021 • 5 minutes to read • Edit Online

The following is a list of the reference content for the Windows application programming interface (API) for desktop and server applications.

Using the Windows API, you can develop applications that run successfully on all versions of Windows while taking advantage of the features and capabilities unique to each version. (Note that this was formerly called the Win32 API. The name Windows API more accurately reflects its roots in 16-bit Windows and its support on 64-bit Windows.)

User interface

The Windows UI API create and use windows to display output, prompt for user input, and carry out the other tasks that support interaction with the user. Most applications create at least one window.

- Accessibility
- Desktop Window Manager (DWM)
- Globalization Services
- High DPI
- Multilingual User Interface (MUI)
- National Language Support (NLS)
- User Interface elements:
  - Buttons
  - Carets
  - Combo Boxes
  - Common Dialog Boxes
  - Common Controls
  - Cursors
  - Dialog Boxes
  - Edit Controls
  - Header Controls
  - Icons
  - Keyboard Accelerators
  - List Boxes
  - List-View Controls
  - Menus
  - Progress Bars
  - Property Sheets
  - Rich Edit Controls
  - Scroll Bars
  - Static Controls
  - Strings

    - Toolbars
    - Tooltips
    - Trackbars
    - Tree-View Controls
- Windows Animation Manager
- Windows Ribbon Framework

Windows environment (Shell)

- Windows Property System
- Windows Shell
- Windows Search
- Consoles

User input and messaging

- User Interaction
    - Direct Manipulation
    - Ink input
    - Input Feedback Configuration
    - Interaction Context
    - Pointer Device Input Stack
    - Pointer Input Messages and Notifications
    - Radial controller input
    - Text Services Framework
    - Touch Hit Testing
    - Touch Injection
- Legacy User Interaction
    - Touch Input
    - Keyboard Input
    - Mouse Input
    - Raw Input
- Windows and Messages:
    - Messages and Message Queues
    - Windows
    - Window Classes
    - Window Procedures
    - Timers
    - Window Properties
    - Hooks

Data access and storage

- Background Intelligent Transfer Service (BITS)
- Data Backup
    - Backup

    - Data Deduplication
    - Volume Shadow Copy
    - Windows Server Backup
- Data Exchange:
    - Clipboard
    - Dynamic Data Exchange (DDE)
    - Dynamic Data Exchange Management (DDEML)
- Directory Management
- Disk Management
- Distributed File System (DFS)
- Distributed File System Replication
- Extensible Storage Engine
- Files and I/O (Local file system)
- iSCSI Discovery Library API
- Offline Files
- Packaging
- Remote Differential Compression
- Transactional NTFS
- Volume Management
- Virtual Hard Disk (VHD)
- Windows Storage Management
- Windows Data Access Components
    - Microsoft Open Database Connectivity (ODBC)
    - Microsoft OLE DB
    - Microsoft ActiveX Data Objects (ADO)

Diagnostics

The Diagnostics API enable you to troubleshoot application or system problems and monitor performance.

- Application Recovery and Restart
- Debugging
- Error Handling
- Event Logging
- Event Tracing
- Hardware Counter Profiling (HCP)
- Network Diagnostics Framework (NDF)
- Network Monitor
- Performance Counters
- Performance Logs and Alerts (PLA)
- Process Snapshotting

- Process Status (PSAPI)
- Structured Exception Handling
- System Monitor
- Wait Chain Traversal
- Windows Error Reporting (WER)
- Windows Event Log
- Windows Troubleshooting Platform

Graphics and multimedia

The Graphics, multimedia, audio, and video APIs enable applications to incorporate formatted text, graphics, audio, and video.

- Core Audio
- Direct2D
- DirectComposition
- DirectShow
- DirectWrite
- DirectX
- Graphics Device Interface (GDI)
- GDI+
- Media Streaming
- Microsoft Media Foundation
- Microsoft TV Technologies
- OpenGL
- Monitor Configuration
- Multiple Display Monitors
- Picture Acquisition
- Windows Color System
- Windows Imaging Component (WIC)
- Windows Media Audio and Video Codec and DSP
- Windows Media Center
- Windows Media Format
- Windows Media Library Sharing Services
- Windows Media Player
- Windows Media Services
- Windows Movie Maker
- Windows Multimedia

Devices

- AllJoyn
- Communications Resources
- Device Access
- Device Management
- Enhanced Storage
- Function Discovery
- Image Mastering

- Location
- PnP-X Association Database
- Printing
  - Print Spooler
  - Print Document Package
  - Print Schema Specification
  - Print Ticket
  - XPS Print
- Sensors
- System Event Notification Service (SENS)
- Tool Help
- UPnP
- Web Services on Devices
- Windows Image Acquisition (WIA)
- Windows Media Device Manager
- Windows Portable Devices

System services

The System Services APIs give applications access to the resources of the computer and the features of the underlying operating system, such as memory, file systems, devices, processes, and threads.

- COM
- COM+
- Compression API
- Distributed Transaction Coordinator (DTC)
- Dynamic-Link Libraries (DLLs)
- Help API
- Interprocess Communications:
  - Mailslots
  - Pipes
- Kernel Transaction Manager (KTM)
- Memory Management
- Operation Recorder
- Power Management
- Remote Desktop Services
- Processes
- Services
- Synchronization
- Threads
- Windows Desktop Sharing
- Windows System Information
  - Handle and Objects
  - Registry
  - Time
  - Time Provider

Security and identity

The Security and Identity APIs enable password authentication at logon, discretionary protection for all sharable system objects, privileged access control, rights management, and security auditing.

- Authentication
- Authorization
- Certificate Enrollment
- Cryptography
- Cryptographic Next Generation (CNG)
- Directory Services
  - Active Directory Domain Services
  - Active Directory Service Interfaces (ADSI)
- Extensible Authentication Protocol (EAP)
- Extensible Authentication Protocol Host (EAPHost)
- MS-CHAP Password Management
- Network Access Protection (NAP)
- Network Policy Server Extensions (NPS)
- Parental Controls
- Security WMI Providers
- TPM Base Services (TBS)
- Windows Biometric Framework

Application installation and servicing

- Games Explorer
- Side-by-side Assemblies
- Packaging, deployment, and query APIs
- Developer License
- Restart Manager
- Windows Installer

System admin and management

The System administration interfaces enable you to install, configure, and service applications or systems.

- Boot Configuration Data WMI Provider
- Failover Clusters
- File Server Resource Manager (FSRM)
- Group Policy
- Microsoft Management Console (MMC) 2.0
- NetShell
- Settings Management Infrastructure
- Software Inventory Logging
- Software Licensing
- Restart Manager
- Settings Management Infrastructure
- System Restore
- System Shutdown
- Task Scheduler
- User Access Logging

- Windows Virtual PC
- Microsoft Virtual Server
- Network Load Balancing Provider
- Windows Defender WMI v2
- Windows Deployment Services
- Windows Genuine Advantage
- Windows Management Infrastructure
- Windows Management Instrumentation (WMI)
- Windows Remote Management
- Windows Resource Protection
- Windows Server Update Services
- Windows System Assessment Tool
- Windows Update Agent

Networking and internet

The Networking APIs enable communication between applications over a network. You can also create and manage access to shared resources, such as directories and network printers.

- Domain Name System (DNS)
- Dynamic Host Configuration Protocol (DHCP)
- Fax Service
- Get Connected Wizard
- HTTP Server
- Internet Connection Sharing and Firewall
- IP Helper
- IPv6 Internet Connection Firewall
- Management Information Base
- Message Queuing (MSMQ)
- Multicast Address Dynamic Client Allocation Protocol (MADCAP)
- Network Address Translation (NAT)
- Network List Manager (NLM)
- Network Management
- Network Share Management
- Peer-to-Peer
- Quality of Service (QOS)
- Remote Procedure Call
- Routing and Remote Access Service (RAS)
- Simple Network Management Protocol (SNMP)
- SMB Management
- Telephony Application Programming Interfaces (TAPI)
- WebDAV
- WebSocket Protocol Component
- Wireless networking:
  - Bluetooth
  - IrDA
  - Mobile Broadband
  - Native Wifi

    - Windows Connect Now
    - Windows Connection Manager
- Windows Filtering Platform
- Windows Firewall with Advanced Security
- Windows HTTP Services (WinHTTP)
- Windows Internet (WinINet)
- Windows Networking (WNet)
- Windows Network Virtualization
- Windows RSS Platform
- Windows Sockets (Winsock)
- Windows Web Services
- XML HTTP Extended Request

Deprecated or legacy APIs

The following are technologies and APIs that are outdated or have been replaced or deprecated from the Windows client and server operating systems.

- DirectMusic
- DirectSound
- Microsoft UDDI SDK is now included with Microsoft BizTalk Server.
- Network Dynamic Data Exchange (DDE)
- Remote Installation Service: Use Windows Deployment Services instead.
- Virtual Disk Service (VDS): Use Windows Storage Management instead.
- Terminal Services: Use Remote Desktop Services.
- Windows Media Rights Manager
- Windows Messaging (MAPI): Use Office MAPI instead.
- Windows Gadget Platform: Create UWP apps instead.
- Windows Sidebar: Create UWP apps instead.
- Windows SideShow: No replacement.
- WPF Bitmap Effects

Windows API sets

1/14/2021 • 3 minutes to read • Edit Online

All versions of Windows 10 share a common base of OS components that is called the *core OS* (in some contexts this common base is also called *OneCore*). In core OS components, Win32 APIs are organized into functional groups called *API sets*.

The purpose of an API set is to provide an architectural separation from the host DLL in which a given Win32 API is implemented and the functional contract to which the API belongs. The decoupling that API sets provide between implementation and contracts offers many engineering advantages for developers. In particular, using API sets in your code can improve compatibility with all Windows 10 devices.

API sets specifically address the following scenarios:

- Although the full breadth of the Win32 API is supported on PCs, only a subset of the Win32 API is available on other Windows 10 devices such as HoloLens, Xbox, and other devices running Windows 10x. The API set name provides a query mechanism to cleanly detect whether an API is available on any given device.
- Some Win32 API implementations exist in DLLs with different names across different Windows 10 devices. Using API set names instead of DLL names when detecting API availability and delay loading APIs provide a correct route to the implementation no matter where the API is actually implemented.

For more details, see API set loader operation and Detect API set availability.

Linking to umbrella libraries

To make it easier to restrict your code to Win32 APIs that are supported in the core OS, we provide a series of *umbrella libraries*. For example, an umbrella library named OneCore.lib provides the exports for the subset of Win32 APIs that are common to all Windows 10 devices.

The APIs in an umbrella library may be implemented across a range of modules. The umbrella library abstracts those details away from you, making your code more portable across Windows 10 versions and devices. Instead of linking to libraries such as kernel32.lib and advapi32.lib, simply link your desktop app or driver with the umbrella library that contains the set of core OS APIs that you're interested in.

For more details, see Windows umbrella libraries.

API set contract names

API sets are identified by a strong contract name that follows these standard conventions recognized by the library loader.

- The name must begin either with the string **api-** or **ext-**.
  - Names that begin with **api-** represent APIs that are guaranteed to exist on all Windows 10 versions.
  - Names that begin with **ext-** represent APIs that may not exist on all Windows 10 versions.
- The name must end with the sequence **l-<n>-<n>-<n>**, where **n** consists of decimal digits.
- The body of the name can be alphanumeric characters, or dashes (-).
- The name is case insensitive.

Here are some examples of API set contract names:

- **api-ms-win-core-ums-l1-1-0**
- **ext-ms-win-com-ole32-l1-1-5**
- **ext-ms-win-ntuser-window-l1-1-0**
- **ext-ms-win-ntuser-window-l1-1-1**

You can use an API set name in the context of a loader operation such as LoadLibrary or P/Invoke instead of a DLL module name to ensure a correct route to the implementation no matter where the API is actually implemented on the current device. However, when you do this you must append the string **.dll** at the end of the contract name. This is a requirement of the loader to function properly, and is not considered actually a part of the contract name. Although contract names appear similar to DLL names in this context, they are fundamentally different from DLL module names and do not directly refer to a file on disk.

Except for appending the string **.dll** in loader operations, API set contract names should be considered an immutable identifier that corresponds to a specific contract version.

Identifying API sets for Win32 APIs

To identify whether a particular Win32 API belongs to an API set, review the requirements table in the reference documentation for the API. If the API belongs to an API set, the requirements table in the article lists the API set name and the Windows version in which the API was first introduced to the API set. For examples of APIs that belong to an API set, see these articles:

- AllowSetForegroundWindow
- FindWindowsEx
- GetClassFile

In this section

- API set loader operation
- Detect API set availability
- Windows umbrella libraries

API set loader operation

11/2/2020 • 2 minutes to read • Edit Online

API sets rely on OS support in the library loader to effectively introduce a module namespace redirection into the library binding process. The API set contract name is used by library loader to perform a runtime redirection of the reference to a target host binary that houses the appropriate implementation of the API set.

When the loader encounters a dependency on an API set at run time, the loader consults configuration data in the image to identify the host binary for an API set. This configuration data is called the **API set schema**. The schema is assembled as a property of the OS, and the mapping between API sets and binaries may differ depending on which binaries are included in a given device. The schema enables an imported function in a single binary to be routed correctly on different devices, even if the module names of the binary host have been renamed or completely refactored on different Windows devices.

Windows 10 supports two standard techniques to consume and interface with API sets: **direct forwarding** and **reverse forwarding**.

Direct forwarding

In this configuration, the consuming code imports an API set module name directly. This import is resolved in a single operation, and is the most efficient method with the least overhead. Conceptually, this resolution may point to different binaries on different Windows 10 devices, as is shown in the following example:

Imported API set: **api-feature1-l1-1-0.dll**

- Windows PC -> **feature1.dll**
- HoloLens -> **feature1_holo.dll**
- IoT -> **feature1_iot.dll**

Because the mappings are kept in a custom schema data repository, it means that an API set name that ends with **.dll** does not directly refer to a file on disk. The **.dll** part of the API set name is only a convention required by the loader. The API set name is more like an alias or a virtual name for a physical DLL file. This makes the name portable across the entire range of Windows 10 devices.

Reverse forwarding

While API set names provide a stable namespace for modules across devices, it is not always practical to convert every binary to this new system. For example, an application may have been in common use for many years, and recompiling the application's binaries may not be feasible. Additionally, some applications may need to continue to run on systems built before specific API sets were introduced.

To accommodate this level of compatibility, a system of *forwarders* are provided on all Windows 10 devices that cover a subset of the Win32 API surface. These forwarders use the module names that were introduced on Windows PCs, and leverage the API Set system to provide compatibility across all Windows 10 devices.

The loader operation beahaves like this:

1. On a device other than a Windows PC, the loader is presented a legacy Windows PC module name dependency that is not present on the device.
2. The loader locates an API set forwarder for this module and loads it into memory.
3. The forwarder has a mapping for the API set for the given function being called.
4. The loader finds the proper host binary for the given device.

Conceptually, the mapping looks like:

Imported DLL: **feature1.dll**

- Windows PC -> **feature1.dll**
- HoloLens -> **feature1.dll** forwarder -> **api-feature1-l1-1-0.dll** -> **feature1_holo.dll**
- IoT -> **feature1.dll** forwarder -> **api-feature1-l1-1-0.dll** -> **feature1_iot.dll**

The end result is functionally the same as direct forwarding, but it accomplishes it in a way that maximizes application compatibility.

> **NOTE**
> Reverse forwarding provides coverage only for a subset of the Win32 API surface. It does not allow applications that target desktop versions of Windows 10 to run on all Windows 10 devices.

Detect API set availability

11/2/2020 • 2 minutes to read • Edit Online

In some cases, a given API set contract name may be intentionally mapped to an empty module name on some Windows 10 devices. The reasons for this vary, but a common example is that an expensive feature in terms of system resources may be removed from the Windows OS when configured for a resource-constrained device. This poses a challenge for applications to gracefully handle optional features at the API level.

The traditional approach for testing whether a Win32 API is available is to use LoadLibrary or GetProcAddress. However, these are not a reliable means for testing API sets because of the reverse forwarding support in Windows 10. When reverse forwarding is applied to a given API, **LoadLibrary** or **GetProcAddress** may resolve to a valid function pointer even in cases where the internal implementation has been removed. In this case, the function pointer will be pointing to a stub function that simply returns an error.

In order to detect this case, you can use the IsApiSetImplemented function to query the underlying availability of a given API implementation. This test validates that calling this function will result in executing a functional implementation of the API.

The following code example demonstrates how to use **IsApiSetImplemented** to determine whether the WTSEnumerateSessions function is available on the current device before calling it.

```
#include <windows.h>
#include <stdio.h>
#include <Wtsapi32.h>

int __cdecl wmain(int /* argc */, PCWSTR /* argv */ [])
{
    PWTS_SESSION_INFO pInfo = {};
    DWORD count = 0;

    if (!IsApiSetImplemented("ext-ms-win-session-wtsapi32-l1-1-0"))
    {
        wprintf(L"IsApiSetImplemented on ext-ms-win-session-wtsapi32-l1-1-0 returns FALSE\n");
    }
    else
    {
        if (WTSEnumerateSessionsW(WTS_CURRENT_SERVER_HANDLE, 0, 1, &pInfo, &count))
        {
            wprintf(L"SessionCount = %d\n", count);

            for (ULONG i = 0; i < count; i++)
            {
                PWTS_SESSION_INFO pCurInfo = &pInfo[i];
                wprintf(L"    %s: ID = %d, state = %d\n", pCurInfo->pWinStationName,
                    pCurInfo->SessionId, pCurInfo->State);
            }

            WTSFreeMemory(pInfo);
        }
        else
        {
            wprintf(L"WTSEnumerateSessions failure : %x\n", GetLastError());
        }
    }

    return 0;
}
```

Windows umbrella libraries

11/2/2020 • 2 minutes to read • Edit Online

An *umbrella library* is a single static-link library that exports a subset of Win32 APIs. For example, an umbrella library named **OneCore.lib** provides the exports for the subset of Win32 APIs that are common to all Windows 10 devices.

The APIs in an umbrella library may be implemented across a range of modules (whether that's an API set or a DLL), but the umbrella library abstracts that detail away from you, making your app more portable across operating system versions. Simply link your desktop app or driver with the umbrella library that contains the set of APIs that you're interested in, and that's all you need to do.

LIBRARY	DESCRIPTION
OneCore.lib	This library provides the exports for the subset of Win32 APIs that are common to all Windows 10 devices. Link your desktop app or driver with OneCore.lib (and no other libraries) to access these APIs. If you link your app or driver to OneCore.lib, and you only call Win32 APIs in that library, then your app or driver will load successfully on all Windows 10 devices.
OneCore_apiset.lib	This library provides the same coverage as OneCore.lib, but uses API set direct forwarding. Note that using this library will be compatible only with the Windows 10 version as specified by the SDK verion you're targeting, or greater.
OneCoreUap.lib	This library provides the exports for the subset of Win32 APIs that are common to all Windows 10 devices that support the Windows Runtime (WinRT). Link your desktop app or driver with OneCoreUap.lib (and no other libraries) to access these APIs. If you link your app or driver to OneCoreUap.lib, and you only call Win32 APIs in that library, then your app or driver will load successfully on all Windows 10 devices that support the UWP.
OneCoreUAP_apiset.lib	This library provides the same coverage as OneCoreUAP.lib, but uses the API set direct forwarding technique. Note that using this library will be compatible only with the Windows 10 version as specified by the SDK you're targeting, or greater.

Calling WinRT APIs from a desktop app

11/2/2020 • 2 minutes to read • Edit Online

The general rule is that a desktop app can call a WinRT API. However, some APIs, including the XAML UI APIs, require the calling app to have a package identity. UWP apps have a well-defined app model, and they have a package identity. The other types of desktop apps have neither a well-defined app model, nor a package identity. Therefore, if an API requires a package identity, a WPF, Windows Forms, or Win32 app cannot call it unless the app is packaged in an MSIX package.

The DualApiPartition attribute

This is the process to follow whenever there's a particular WinRT that you'd like to call from your desktop app. This process will answer the question of whether the API is allowed to be called from a desktop app. First, visit the Windows API reference for Windows Runtime apps, find the reference topic for the class or member API you're interested in, and check whether the **DualApiPartition** attribute is listed in the Attributes section.

If the DualApiPartition attribute is listed

If the DualApiPartition attribute is listed, the API does not need the calling app to have a package identity, and the API is allowed to be called from any desktop app. **Windows.Devices.Geolocation.Geolocator** is an example; an app does not need to be uniquely identified in order to perform location tasks.

If the DualApiPartition attribute is not listed

If the DualApiPartition attribute is not listed, the API does need the calling app to have a package identity. Therefore a WPF, Windows Forms, or Win32 app is not allowed to call the API unless the app has been converted to a UWP app. **Windows.UI.Xaml.Controls.Button** is an example.