INTERNET COMMUNICATION WITH WINDOWS APIS

The Internet **🌐** changed computing by turning isolated machines into a single, global network. Early systems like dial-up services and closed email networks worked, but they were limited, text-only, and cut off from each other.

Modern Internet access fixed that. With one connection and open protocols, users can communicate globally, browse the web, and share data across systems. The World Wide Web added structure, multimedia, and interaction, making information easier to access and use.

On Windows, Internet functionality is exposed through APIs that handle networking and data exchange. Two key ones are **WinSock** and **WinInet**.

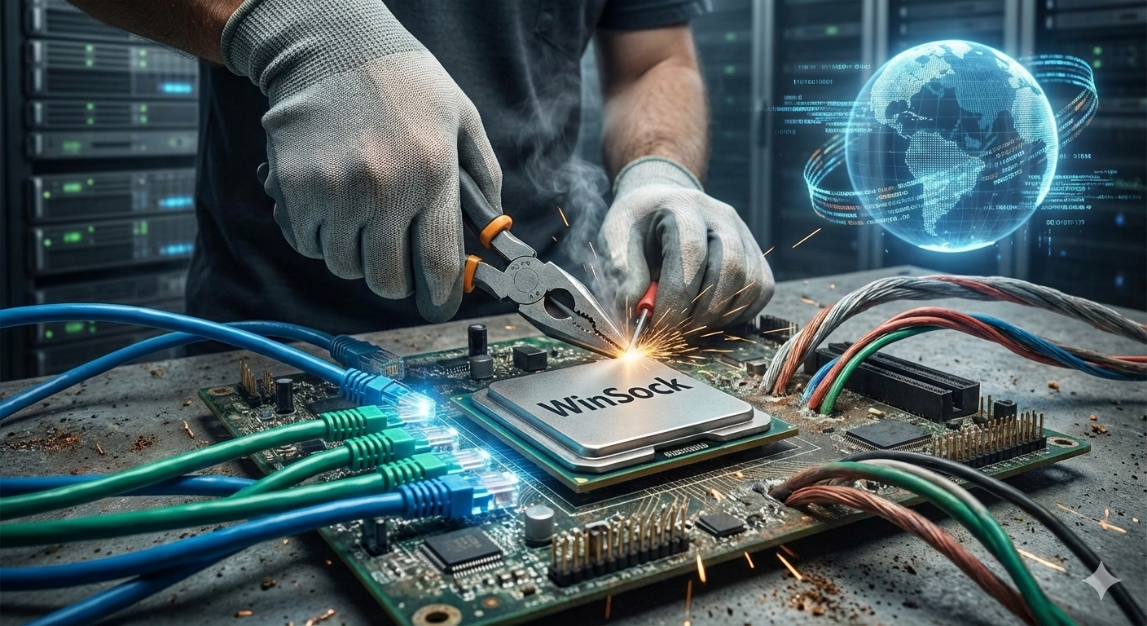
**🧠** WinSock: Low-Level Networking

WinSock (Windows Sockets) is the foundation of Internet programming on Windows.  
It gives direct control over sockets, connections, data transfer, and network errors.

Use WinSock when you need:

* Custom protocols
* Fine-grained control over data
* Full ownership of the network logic

It’s powerful, but you’re closer to the metal.



**🌍** WinInet: High-Level Internet Tasks

WinInet builds on WinSock and handles common Internet operations for you, like:

* HTTP requests
* FTP file transfers
* Basic web access

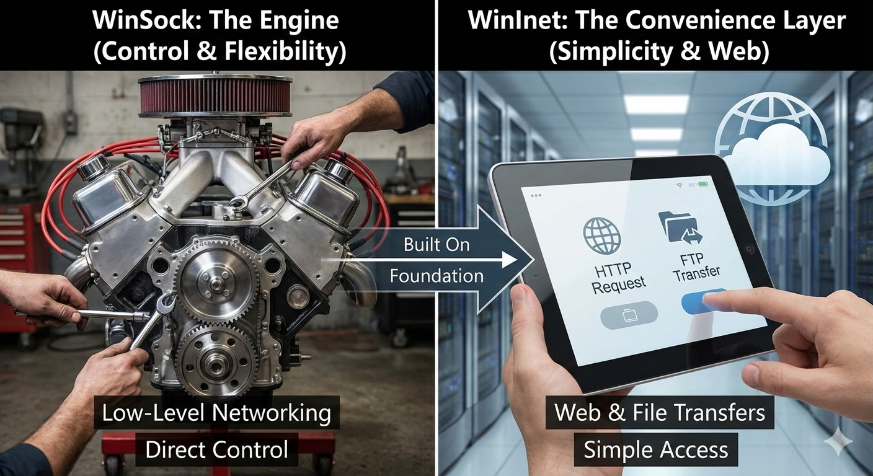
It’s ideal when you just want to download data, upload files, or talk to web services without managing raw sockets.



**🎯** Choosing the Right API

* **WinSock** → control, flexibility, low-level networking
* **WinInet** → simplicity, web and file transfers

In short: WinSock is the engine. WinInet is the convenience layer.



**🔌** Windows Sockets (WinSock): The Basics

Windows Sockets (WinSock) is the core API for network programming on Windows. It provides a standard way for applications to communicate over local networks and the Internet, forming the foundation for most Windows networking software.

**🧩** The Socket Concept

Sockets originated from the Berkeley UNIX networking model and became the standard interface for network communication across operating systems.  
WinSock is Microsoft’s implementation of this model on Windows.

A **socket** represents one endpoint of a network connection.

**🌐** TCP/IP and How Sockets Fit In

Sockets operate on top of the TCP/IP protocol stack, which has two key layers:

* **IP (Internet Protocol)**  
  Handles addressing and routing. It breaks data into packets, assigns destination addresses, and moves them across networks.
* **TCP (Transmission Control Protocol)**  
  Provides reliable communication. It establishes a connection, guarantees ordered delivery, and retransmits lost data.

Together, TCP/IP allows applications to exchange data reliably across the Internet.

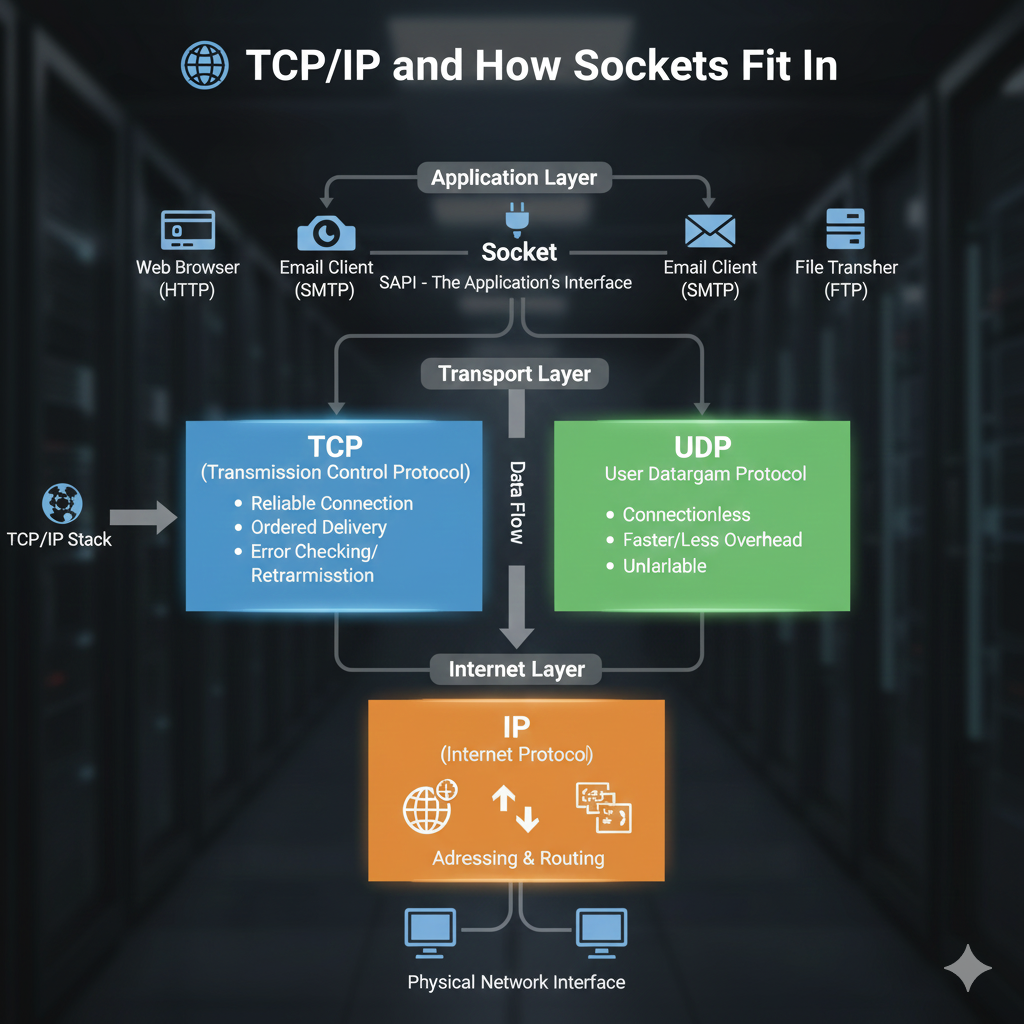
**🎯** Socket Addresses: Identifying Endpoints

Each socket is uniquely identified by:

* **IP address** – identifies the machine (e.g., 209.86.105.231)
* **Port number** – identifies the specific service or application

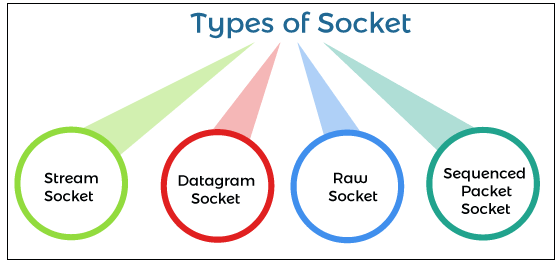
IP = *which computer*  
Port = *which program on that computer*

This combination defines a communication endpoint.



Types of Sockets & Connection Setup

WinSock supports **four socket types**. Two are common, two are specialized.



**🧵** Stream Sockets (SOCK\_STREAM)

* **Connection-oriented**
* **Reliable, ordered delivery**
* Built on **TCP**
* Think: phone call — steady, guaranteed data flow
* Used by: web browsers, email, file transfers

**📦** Datagram Sockets (SOCK\_DGRAM)

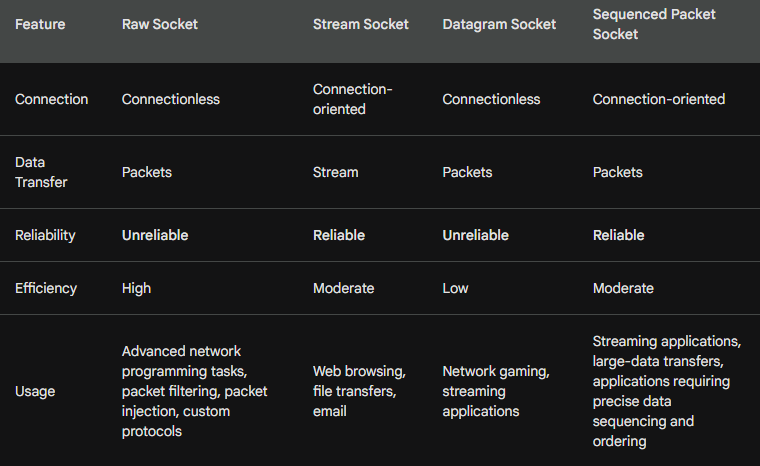
* **Connectionless**
* **Unreliable**, no delivery guarantees
* Built on **UDP**
* Data sent as independent packets
* Used where **speed > reliability** (games, DNS, streaming)

**🧪** Raw Sockets (Advanced)

* Direct access to **IP packets**
* You build headers + payload yourself
* Used for:
  + Packet sniffing
  + Packet injection
  + Custom protocols
* **Low-level, dangerous if misused**
* Mostly for diagnostics, security, research

**📑** Sequenced Packet Sockets (Rare)

* **Connection-oriented**
* Preserves **packet boundaries**
* Reliable *and* ordered
* Middle ground between stream and datagram
* Rarely used in WinSock apps today



Establishing a Connection (Typical Flow)

1. **Create a socket**  
   Choose type (stream, datagram) and protocol.
2. **Bind the socket**  
   Assign IP address + port.
3. **Connect or Listen**
   * Clients → connect()
   * Servers → listen() then accept()
   * Datagram sockets usually skip connection setup

WinSock Functions & Structures **🛠**

WinSock provides APIs for:

* Creating, binding, closing sockets
* Sending and receiving data
* Handling addresses and ports
* Detecting network errors

Key structures:

* **sockaddr** – holds IP + port
* **WSADATA** – WinSock version and system info

**🌍** Where WinSock Is Used

* Web browsers
* Email clients
* File transfer tools
* Online games
* Remote access software

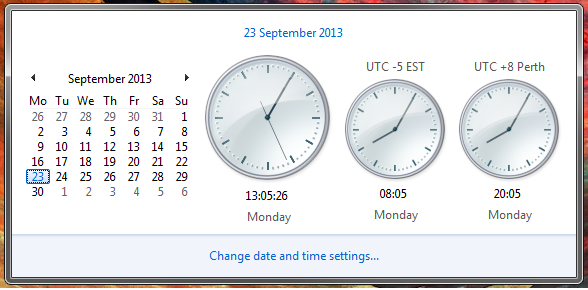
Accurate Time via Network Time Services

Accurate time matters for databases, logs, security checks, synchronization, and distributed systems.  
Local system clocks drift over time due to imperfect hardware oscillators and software issues. Network time services exist to correct this drift by synchronizing systems to authoritative time sources.

**🌐** Network Time Services

Instead of trusting a local clock, a machine periodically asks a trusted server for the current time.  
These servers are maintained by organizations with access to atomic clocks (e.g., **NIST**).

Time is exchanged using standardized network protocols so any system can participate.



**🧭** NIST Network Time Service (NTFS)

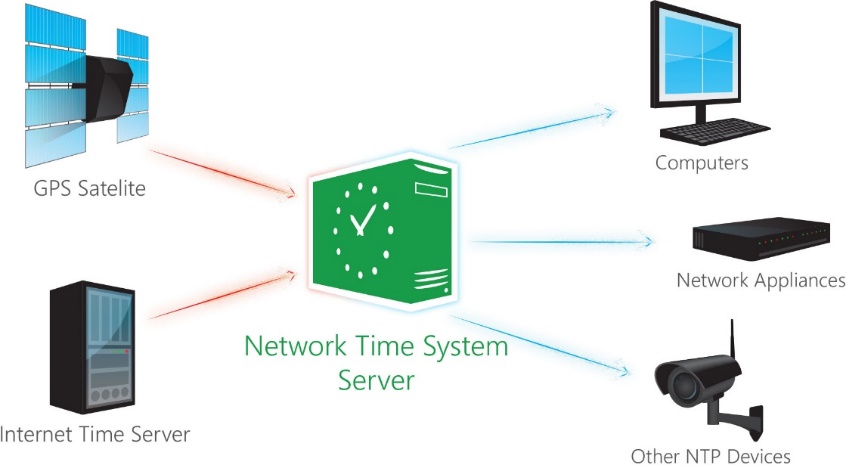
NIST provides public time servers reachable over the Internet.

* Uses the **Time Protocol (RFC 868)**
* Server port: **37**
* Data returned:
  + A **32-bit integer**
  + Number of seconds since **January 1, 1900 (UTC)**

Client flow:

1. Connect to a NIST time server
2. Read the 32-bit time value
3. Convert it to a usable date/time
4. Adjust the local clock or use it as a reference

Simple. No negotiation. No packets back and forth.



**⏳** Why Local Clocks Drift

Local clocks rely on **quartz oscillators**, which:

* Change frequency with temperature
* Degrade with age
* Vary slightly between devices

Software can worsen this:

* Timer bugs
* Scheduler delays
* Misconfigured clock adjustments

Even small errors accumulate into **seconds or minutes over time**.

**🛠** How Network Time Reduces Drift

Network time services correct drift by:

* **Regular synchronization**  
  Periodic updates pull the clock back to the correct time.
* **High-precision reference**  
  Servers are synchronized to atomic clocks.
* **Reduced reliance on local hardware**  
  The system clock becomes an approximation, not the authority.

Network latency introduces small errors, but these are far smaller than long-term drift from local clocks.

**✅** Benefits of Network Time

* Accurate timestamps
* Consistent logs across systems
* No manual clock setting
* Better coordination in distributed systems

This whole section boils down to:

*Local clocks drift. Ask someone smarter for the time.* *That’s it.*

NETTIME Program — What It *Really* Does

NETTIME is a simple Windows program that **sets your system clock using an Internet time server**.

That’s it. Nothing else.

NETTIME connects to a NIST time server over TCP, reads a 32-bit timestamp (seconds since Jan 1, 1900 UTC), converts it into Windows time formats, and updates the system clock using SetSystemTime.

**🔌** Network Side (WinSock, stripped down)

The networking flow is standard WinSock:

1. **WSAStartup** – initialize WinSock
2. **socket** – create a TCP socket (AF\_INET, SOCK\_STREAM)
3. **connect** – connect to the NIST server (port 37)
4. **recv** – read the 32-bit time value
5. **closesocket / WSACleanup** – clean up

Asynchronous I/O (WSAAsyncSelect) is used so the UI doesn’t freeze while waiting for the server.

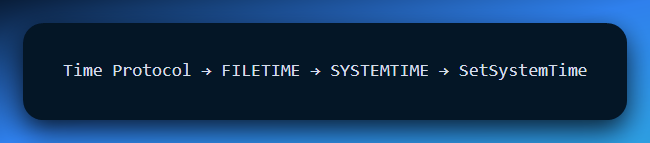
**🕒** Time Conversion (Why it’s a bit annoying)

The server sends: **32-bit seconds since Jan 1, 1900 (UTC)**

Windows wants:

* **SYSTEMTIME**
* via **FILETIME** (100-ns ticks since Jan 1, 1601)

So, NETTIME does:

   
Annoying, but necessary.

**🖥** UI Structure   
Uses a **modeless dialog**

Shows status messages

Buttons:

* Select server
* Sync time
* Exit

The dialog exists mainly to receive socket notifications and show feedback.

**⚠️** Error Handling

If something fails (socket, connection, recv, or SetSystemTime):

* An error message is displayed
* Clock is left unchanged

**🧾** Why MSG\_PEEK Is Used

**recv(... MSG\_PEEK)** reads the incoming data **without removing it** from the socket buffer.  
This allows the program to inspect the data safely before final processing.

NETTIME is basically:

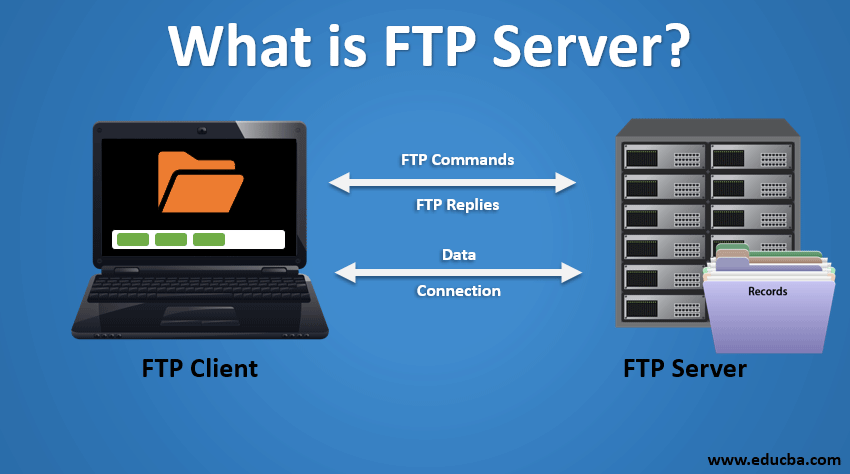
WinSock client + time conversion + SetSystemTime

It’s not about networking tricks — it’s a **teaching example** showing:

* TCP client basics
* Async sockets in a GUI app
* Converting external data into Windows system formats

That’s all. Opens one socket, reads 4 bytes and sets the clock.

WinInet and FTP — High-Level Internet Access on Windows



WinInet is a **high-level Windows API** designed to make Internet communication easy. Instead of dealing with raw sockets and protocol details, WinInet exposes Internet operations in a way that feels similar to standard Windows file I/O.

It supports multiple Internet protocols, most notably:

* **HTTP**
* **FTP**
* **Gopher** (legacy)

The key idea is simple: *WinInet lets you treat Internet resources almost like files.*

**🤔** Why WinInet Exists

Using raw WinSock means manually handling:

* Protocol rules
* Data framing
* Error states
* Authentication
* Connection management

WinInet hides all of that.

With WinInet, you focus on:

* Opening connections
* Reading and writing data
* Navigating directories
* Handling files

Not packets.

**📁** FTP Basics

FTP (File Transfer Protocol) is used to transfer files between computers over a network.

It supports:

* **Anonymous FTP** (no login required)
* **Authenticated FTP** (username + password)

FTP operates in two modes:

* **Active mode** — client initiates data connection
* **Passive mode** — server provides connection details (better for firewalls)

WinInet supports both modes transparently.

**🔧** FTP via WinInet (What You Actually Do)

The typical WinInet + FTP flow looks like this:

1. **InternetOpen**  
   Create an Internet session.
2. **InternetConnect**  
   Connect to an FTP server (anonymous or authenticated).
3. **FTP operations**  
   Download, upload, delete files, or manage directories.
4. **Close handles**  
   Clean up when done.

That’s it.

**📦** Common FTP Operations (WinInet)

WinInet provides direct functions for common FTP tasks:

* **Download files**  
  FtpOpenFile, FtpReadFile
* **Upload files**  
  FtpCreateFile, FtpWriteFile
* **Directory management**  
  FtpCreateDirectory, FtpRemoveDirectory, FtpSetCurrentDirectory
* **File management**  
  FtpDeleteFile, FtpGetFileAttributes
* **Raw commands**  
  FtpSendCommand (when you need fine control)

**🧪** UPDDEMO Program (What It Demonstrates)

The **UPDDEMO** sample program shows WinInet FTP in action.

What it does:

* Connects to an **anonymous FTP server**
* Navigates to a directory
* Downloads a file
* Saves it locally

The program demonstrates how WinInet turns FTP into a **simple, readable sequence of function calls**, without exposing protocol complexity.

**🎯** Why This Section Matters

This is the first point in the book where:

* Networking becomes **practical**, not theoretical.
* Internet access looks **Windows-native.**
* You see how **real applications** download updates and files.

WinSock teaches *how networks work.*  
WinInet teaches *how apps actually use them.*

**🔚** Bottom Line

* **WinSock** → low-level, flexible, complex
* **WinInet** → high-level, simple, practical

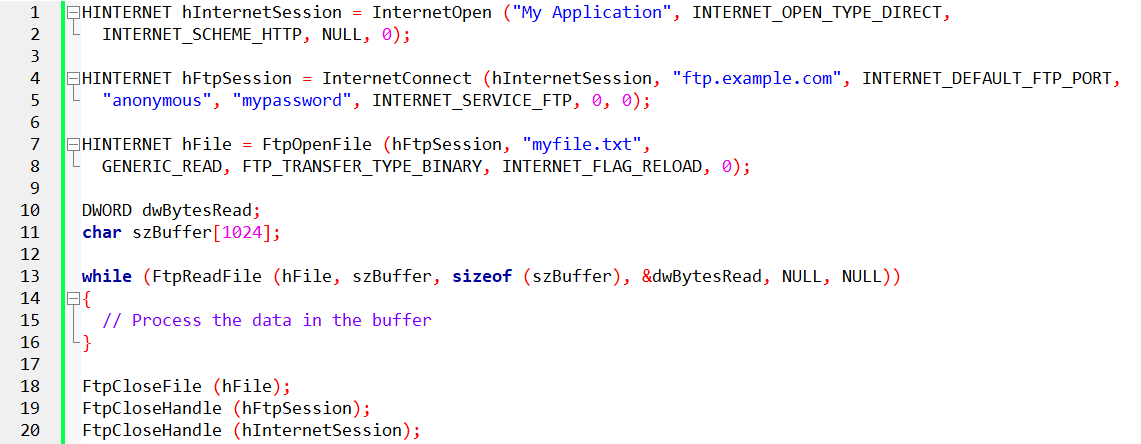
If you want control → WinSock  
If you want results → WinInet

Using the FTP API

To use the FTP API, you will need to include the WININET.H header file.

You will also need to link with the WININET.LIB library.

The following is how to download a file from an FTP server using the FTP API:



This code will download the file myfile.txt from the FTP server ftp.example.com.

The file will be saved to the current directory on the local machine.

The FTP API is a powerful tool that can be used to download files, upload files, manage directories, and send commands to FTP servers.

It’s a useful tool for developers who need to access FTP servers from their applications.

Deleting Files

Use the FtpDeleteFile function.

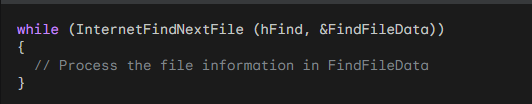
This function takes two arguments: the FTP session handle and the file name.

The file name must be a fully qualified path name, including the server name.



This function returns a handle to the first file that matches the search pattern.

You can use this handle to call the InternetFindNextFile function to get the names of additional matching files.



When you are finished searching for files, you should call the InternetCloseHandle function to close the handle.

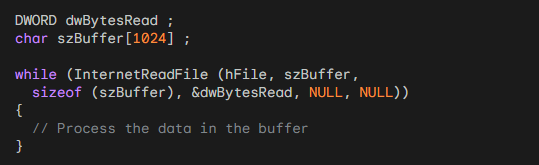


Opening and Reading Files

To open a file on an FTP server, you can use the FtpFileOpen function. This function takes four arguments: the FTP session handle, the file name, the file access mode, the transfer type, and the flags.



Once you have opened a file, you can read data from it using the InternetReadFile function.



When you are finished reading from a file, you should call the FtpCloseFile function to close the handle.



UPDDEMO.C

This chapter concludes with a practical, real-world example: **UPDDEMO**, a Windows application that uses **WinInet + FTP** to check for and download updates. This isn’t theory — it’s a full pipeline: UI → network → disk → feedback.

**🧩** Main Window Logic (Message-Driven Core)

The **window procedure** is the backbone of the application. It reacts to user actions and system events:

* **WM\_CREATE**  
  Initializes the window state and prepares the scroll bar.
* **WM\_SIZE**  
  Recalculates layout and scroll ranges when the window changes size.
* **WM\_VSCROLL**  
  Handles scrolling through the file list.
* **WM\_USER\_CHECKFILES**  
  Triggered when the user checks for updates.  
  If updates are missing, the download dialog is shown.
* **WM\_USER\_GETFILES**  
  Reads the downloaded files from disk and displays them.
* **WM\_PAINT**  
  Draws the visible file list.
* **WM\_DESTROY**  
  Final cleanup. No leaks, no leftovers.

This design keeps the UI responsive while delegating heavy work elsewhere.

**🧾** Dialog Procedure (User Control Layer)

The **dialog procedure** manages the download UI:

* **WM\_INITDIALOG**  
  Initializes controls and launches the FTP worker thread.
* **WM\_COMMAND**  
  Handles user actions.  
  If *Cancel* is pressed, the FTP thread is stopped safely.

The dialog doesn’t do the work — it **controls** the work.

**🌐** FtpThread — The Real Worker

All network activity happens in a **separate thread**, preventing UI freezes.

**Execution flow:**

1. Open Internet session
2. Connect to FTP server
3. Change to target directory
4. Enumerate matching files
5. Download files to disk
6. Close FTP session
7. Close Internet session

This separation is critical: **UI stays alive, network stays isolated**.

**📂** GetFileList — Local File Processing

Once downloads are complete, files are processed locally:

* Locate files matching the template
* Read file size
* Allocate memory
* Load filenames and contents
* Close file handles
* Sort files alphabetically

This prepares the data for display and verification.

**🔁** Compare — Sorting Logic Made Explicit

The **Compare** function exists for one reason: **ordering**.

Uses lstrcmp to compare filenames

Returns:

* -1 → comes before
* 0 → equal
* 1 → comes after

Passed directly into qsort, it ensures files appear in clean alphabetical order — no hacks, no custom loops.

**🔘** ButtonSwitch — Finishing Touch

This function finalizes the user experience:

Displays the final status:

* Error → download failed or canceled
* Success → files downloaded correctly

Converts **Cancel** → **OK**

Updates the button ID to IDOK

This small detail matters — it cleanly signals *completion* and restores user control.

**💥** Chapter Takeaway

This chapter wasn’t about FTP alone.

It demonstrated:

* Message-driven Windows design
* Threaded networking
* Safe UI + worker separation
* Real file handling
* Clean shutdown paths

This is **classic WinAPI architecture** done right.

*We’re done guys!!!! Done!!! Now it’s up to you to practice*👍*Best of luck in your endeavours! You are now well equipped to tackle operating systems books by Abraham Silberschatz which implements what we’ve learnt about WinAPI! Bye!*

