AI Drift

Full Technical Specification: Thunking Mechanisms in Windows (16-bit → 32-bit)

Document version: 1.0 **Applicable to:** Windows 3.x, Windows 95/98/Me, Windows NT 3.1–4.0
Author: System Architecture Expert

Table of Contents

Introduction: The mixed-bitness problem

16-bit Windows versions 1.x and 2.x ran on Intel 8086/8088 in **real mode**, where a far pointer was a 16-bit segment value (shifted left by 4) plus a 16-bit offset, resulting in a 20-bit physical address. A significant change came with the **Windows/386** family (versions 2.03 and later in 1987), which first introduced a **protected mode** environment for 386 processors[reference:0]. This “386 Enhanced Mode” allowed the system to run multiple MS-DOS applications concurrently in extended memory, moving beyond the 640KB barrier. It was during this period that a far pointer evolved into a **selector:offset** pair, where the selector was an index into a descriptor table[reference:1].

32-bit systems (Windows NT, 95) introduced the **flat model**: a 32-bit linear address (0:32), with `int` size = 32 bits, stack SS:ESP, and support for up to 4GB of virtual address space.

Direct calling between 16-bit code (real or protected mode) and 32-bit flat-model code is impossible because of:

- different pointer formats (16:16 far pointer vs 0:32 linear),
- different sizes of basic types (16-bit int vs 32-bit int),
- different stack conventions (SS:SP vs SS:ESP),
- the need to switch processor decoding mode (D-bit in segment descriptors for protected mode, or switching between real and protected modes).

A **thunk** is a piece of code (or a set of functions) that transparently transforms a call between these incompatible execution environments.

However, the concept of thunking is much older and broader than the 16-to-32-bit transition. Already

in the real-mode era, Windows faced two fundamental problems that thunks were invented to solve:

1. **Segment swapping (overlays)** – In real mode, code segments could be discarded or moved to disk to save memory. When a far call was made to a function in a potentially absent segment, a mechanism was needed to check presence, reload the segment from disk if necessary, and then transfer control. This was performed by a **call thunk** (or **instance thunk**). The Windows API function `MakeProcInstance` created such a thunk dynamically, binding a far pointer to a specific instance's data segment (loading `hInstance` into AX before jumping to a system reload thunk).
2. **Return from a discarded segment** – After a function returned, the return address (CS:IP) on the stack might point to a code segment that had been swapped out. To handle this gracefully, Windows used a **return thunk**. The undocumented `CallProcInstance` function acted as such a thunk, checking the presence of the target segment, reloading it if needed, and then jumping to the original return address. This mechanism allowed cooperative multitasking without an MMU, relying on software-managed segment validity.

Thus, even in pure 16-bit Windows, thunks were essential for memory management and for correctly routing callbacks (window procedures, timer procs, etc.) to the appropriate instance's data segment. Later, as 32-bit flat model emerged, the same core idea — a small piece of glue code that translates between different calling conventions, pointer formats, and memory models — was extended to bridge 16-bit and 32-bit code, giving birth to Generic Thunk (NT), Universal Thunk (Win32s), Flat Thunk (Windows 95), and the low-level Thunklet building blocks.

This document describes all these thunk variants in technical detail, from the early real-mode thunks to the last 16-32 bridges used in Windows 9x and NT.

1. Instance Thunk (Win16)

1.1 Architectural root of the problem: shared code, private data

In 16-bit Windows, multiple instances of the same application share code but each instance has its own **data segment (DS)**. When Windows calls a callback function (window procedure, `EnumFonts`, `SetTimer`, etc.), the system does not know which DS to use for that particular instance. The 8086 processor had no MMU; all memory was physical, with no indirection layer. This meant that if the memory manager moved a data segment, it had to know about every reference to it in order to update pointers. This situation gave rise to a special mechanism – the **Instance Thunk**.

1.2 Role of the loader and the NE format

The foundation of the mechanism is laid when the executable file is loaded. Windows 3.x uses the **NE (New Executable)** format, in which each far (FAR) function has a 6-byte record in the Entry Table. This table is placed by the loader in a fixed overhead segment that is shared by all instances of the program.

During loading, Windows **expands each 6-byte entry table record into an 8-byte fragment of machine code** called a **reload thunk**. These thunks become the official, single entry point for calling any corresponding far function.

1.3 Reload thunk: detailed structure and swapping mechanism

Full reload thunk structure (8 bytes):

```
SAR BYTE PTR CS:[xxxx], 1  ; 3 bytes: shift access counter right
INT 3Fh                    ; 2 bytes: software interrupt call
db entry_segment           ; 1 byte : index into the module's segment table
dw entry_offset            ; 2 bytes: function offset inside the segment
```

Each field has a precise purpose:

* **`SAR BYTE PTR CS:[xxxx], 1`** – software “accessed bit” for the **LRU (Least Recently Used)** discarding algorithm. The access counter for the segment is initialised to 1. Every call shifts it right, turning it to 0. Every 250 ms Windows scans the counters and builds an LRU list. If a counter remains 1 (no calls), the segment is a candidate for discarding. This allowed software emulation of an MMU. * **`INT 3Fh`** – system interrupt that invokes the segment loader. The interrupt handler reads the operands that follow the `INT 3Fh` instruction. * **`db entry_segment`** – index of the entry table record for this function in the NE header. It corresponds directly to the `entry_segment` parameter in the NE format. * **`dw entry_offset`** – offset inside the 16-bit code segment, relative to the module start. The loader uses it to compute the final 32-bit linear address after the segment has been loaded.

Two states of a reload thunk:

State 1 – code segment not in memory: The thunk executes SAR, then `INT 3Fh` with its operands. The `INT 3Fh` handler reads `entry_segment` and `entry_offset`, locates the segment in the module's segment table, loads the segment from disk, and updates the table.

State 2 – code segment is in memory: The kernel modifies the reload thunk:

```
SAR BYTE PTR CS:[xxxx], 1  ; 3 bytes — always executed
JMP ssss:oooo              ; 5 bytes — direct jump to the resident function
```

The `JMP` instruction occupies the 5 bytes that previously held `INT 3Fh` and its operands. The address `ssss:oooo` is the real 32-bit linear address of the function.

Life cycle:

1. **Initialisation** – the loader builds the reload thunk from the NE file's Entry Table data. 2. **First call** – SAR executes, then `INT 3Fh`. The handler loads the code segment from disk. 3. **Patching** – after the segment is loaded, the loader overwrites `INT 3Fh` and the next 3 bytes with `JMP ssss:oooo`. 4. **Subsequent calls** – SAR executes, then `JMP` directly to the function (no loading). 5. **Discarding** – when the system needs memory, the kernel may discard the code segment. Upon discarding, the reload thunk is restored to its original state (`INT 3Fh` with operands) using saved data in the segment table.

1.4 Three prolog types for exported functions

The loader not only creates reload thunks, but also **modifies the prolog of exported functions**, replacing the first 2-3 bytes with `NOP` instructions. There are three prolog types:

Type 1 – Load DS from AX (classic, for EXE callbacks) - Works with MakeProcInstance. The loader replaces the first 3 bytes with NOPs, making the prolog dependent on the value in AX.

```
<code>
nop
nop
nop
push bp
mov bp, sp
push ds
mov ds, ax           ; load DS from AX
</code>
```

Type 2 – Load DS from SS (alternative for EXE) - Assumes SS already contains the correct DS. MakeProcInstance is not needed.

```
<code>
mov ax, ss
push bp
mov bp, sp
push ds
mov ds, ax
</code>
```

Type 3 – Load DS from a hard-coded value (for DLLs) - A DLL has only one data instance, its data selector is known at load time. The loader replaces the placeholder ???? with the actual value. MakeProcInstance is not needed.

```
<code>
mov ax, ????          ; actual DLL data selector
push bp
mov bp, sp
push ds
mov ds, ax
</code>
```

1.5 MakeProcInstance: dynamic binding to an instance

MakeProcInstance creates a thunk that binds a function call to a specific instance's data. It dynamically generates an 8-byte code fragment in a fixed memory area:

```
mov     ax, hInstance       ; B8 xx xx  — load the instance's data selector
into AX
jmp     far ptr reload_thunk ; EA xx xx xx xx — jump to the system reload
thunk
```

The second operand of the JMP instruction is the address of the target function's **reload thunk**. The address of this generated **instance thunk** is returned by MakeProcInstance as a FARPROC.

Every such dynamically created thunk must eventually be freed by `FreeProcInstance`. If the instance's data segment is moved in memory, the kernel updates all corresponding thunks by replacing the immediate operand of the `MOV AX, hInstance` instruction (real mode had no indirection; all addresses were physical).

The call chain:

1. **Instance thunk** loads `hInstance` into AX. 2. It jumps to the system **reload thunk**. 3. **Reload thunk** updates the SAR counter; if necessary it loads the segment via `INT 3Fh` (using `entry_segment` and `entry_offset`). 4. **Reload thunk** jumps to the modified prolog of the exported function. 5. The prolog (with the NOPs) saves the old DS and loads the new DS from AX, which was set by the instance thunk. 6. The function body executes.

1.6 CallProcInstance and the return thunk

`CallProcInstance` is an **undocumented kernel function** that, together with a special **return thunk** mechanism, solves the problem of returning into a discarded code segment. Prototype:

```
LONG FAR PASCAL CallProcInstance(HWND hWnd, WORD wMsg, WORD wParam, LONG
lParam);
```

It was used internally by dispatchers such as `CallWindowProc` to invoke a thunk whose address was passed via the **ES:BX** register pair. However, its main role is in the return thunk.

Return thunk mechanism:

- For every discardable code segment, the system creates **one shared return thunk**, pre-placed in the segment's overhead data. - When the kernel discards a code segment, a **stack patcher** walks through the stack of every thread and **replaces** the original return address (`CS:IP`) with the address of that segment's return thunk. - The original offset (`IP`) is saved in the stack location that previously held the caller's DS. - The return thunk is **idempotent**: it can be safely called even if the target segment is already in memory. If the code is already present, it simply restores the original state and jumps to it; if not, it loads the segment. - This idempotency was provided by `CallProcInstance`.

1.7 Evolution and redundancy of MakeProcInstance

Later it was discovered that `MakeProcInstance` was often unnecessary.

* **`loadds for DLLs — because a DLL has only one data instance, the compiler can hard-code the fixed hInstance value directly into the function prolog (type 3). This made MakeProcInstance redundant. * export` for EXEs** – the instance handle can be obtained directly from the stack selector (SS) (type 2), also eliminating the need for `MakeProcInstance` in most applications.

The discovery that the entire work of `MakeProcInstance` was superfluous was made by **Michael Geary**. His `FixDS` technique worked perfectly already in Windows 1.0, and the long-standing practice of using EXPORTS and `MakeProcInstance` turned out to be an unnecessary adventure.

In modern 32-bit and 64-bit Windows, `MakeProcInstance` is a stub macro that simply returns the

passed pointer, and `FreeProcInstance` does nothing.

1.8 Complete call flow summary

1. **Loading** – the loader reconstructs the Entry Table into reload thunks (SAR, `INT 3Fh`, `entry_segment`, `entry_offset`) and modifies the prologs of exported functions (replacing the beginning with NOPs). 2. **Instance thunk creation** – the application calls `MakeProcInstance` with a function pointer and `hInstance`. The kernel generates the thunk `MOV AX, hInstance; JMP reload_thunk`. 3. **Call** – Windows calls the address returned by `MakeProcInstance`. 4. **Instance thunk** – loads `hInstance` into AX and jumps to the reload thunk. 5. **Reload thunk** – executes SAR (updating the LRU counter). If the code segment is absent, `INT 3Fh` uses `entry_segment` and `entry_offset` to load the segment from disk. After loading, `INT 3Fh` is replaced by JMP. 6. **Transfer to function** – the reload thunk jumps to the modified prolog of the exported function. 7. **Prolog** – saves the old DS and loads the new DS from AX (set by the instance thunk). 8. **Function execution**. 9. **Return** – if the code segment was discarded while the function was running, the return thunk mechanism (part of `CallProcInstance`) reloads the segment and transfers control to the saved return address.

1.9 API and ordinals

Function	Export	Ordinal (KERNEL.EXE 3.10)
`MakeProcInstance`	by name	51 (unofficial)
`FreeProcInstance`	by name	**52** (confirmed)
`CallProcInstance`	by name	missing (undocumented)

FreeProcInstance (ordinal 52) removes the instance thunk from internal kernel tables and frees its memory.

2. Generic Thunk (Windows NT)

2.1 WOW architecture in NT

In Windows NT, all 16-bit applications run inside a single process **NTVDM.EXE** (NT Virtual DOS Machine). Each 16-bit application is a **separate thread** inside NTVDM. They all share a single address space but have different TDBs (Task Database). 32-bit DLLs are loaded directly into the NTVDM process address space. Thanks to this, 16-bit code can call 32-bit functions via Generic Thunk.

2.2 KRNL386.EXE exports (ordinals 513–517)

The 16-bit library `KRNL386.EXE` (shipped with Windows NT) added new exports:

Ordinal	Export name	Purpose
513	`LoadLibraryEx32W`	Loads a 32-bit DLL into NTVDM space.
514	`FreeLibrary32W`	Unloads a 32-bit DLL.
515	`GetProcAddress32W`	Returns linear address (0:32) of an exported function.

Ordinal	Export name	Purpose
516	GetVDMPointer32W	Converts a 16:16 pointer to a 32-bit linear address.
517	CallProc32W	Calls a 32-bit function with parameter conversion (Pascal calling convention).

Later CallProcEx32W appeared (no fixed ordinal), supporting __cdecl and the flag CPEX_DEST_CDECL.

2.3 API functions: prototypes and parameters

''LoadLibraryEx32W''

```
HINSTANCE32 LoadLibraryEx32W(LPSTR lpszFile, HFILE hFile, DWORD dwFlags);
// lpszFile – DLL name (ANSI)
// hFile – unused (0)
// dwFlags – can be DONT_RESOLVE_DLL_REFERENCES (0x00000001)
// returns 32-bit handle (HINSTANCE32)
```

''FreeLibrary32W''

```
BOOL FreeLibrary32W(HINSTANCE32 hInst);
// hInst – handle from LoadLibraryEx32W
```

''GetProcAddress32W''

```
FARPROC GetProcAddress32W(HINSTANCE32 hInst, LPCSTR lpszProc);
// lpszProc – function name or ordinal if HIWORD(lpszProc)==0
// returns linear address (0:32)
```

''GetVDMPointer32W''

```
DWORD GetVDMPointer32W(LPVOID lpAddress, UINT fMode);
// lpAddress – 16:16 pointer (selector:offset)
// fMode – 1 = protected mode selector:offset; 0 = real-mode segment:offset
// returns 32-bit linear address or NULL
```

''CallProc32W'' (Pascal)

```
DWORD CallProc32W(FARPROC lpFunction, DWORD dwAddrConvert, DWORD dwParams,
...);
// lpFunction – linear address from GetProcAddress32W
// dwAddrConvert – bitmask (LSB = 1st parameter)
// dwParams – number of parameters (0..32)
```

```
// ... — parameters (each must be DWORD)
```

''CallProcEx32W'' (cdecl)

```
DWORD FAR CDECL CallProcEx32W(DWORD nParams, DWORD fAddressConvert,
                              DWORD lpProcAddress, ...);
// nParams — low bits = parameter count; high bit (0x80000000) =
CPEX_DEST_CDECL
// fAddressConvert — bitmask
// lpProcAddress — linear address
```

2.4 Bitness switching: 66h/67h prefixes and far call

On x86 processors (386+), each code segment has a **D** bit in its descriptor: * D=0 → 16-bit decoding (default IP, SP, operands are 16-bit) * D=1 → 32-bit decoding

Prefix **66h** (operand size override) temporarily toggles operand size to the opposite. Prefix **67h** (address size override) toggles address size.

The code inside `CallProc32W` (executed in a 16-bit segment, D=0) looks like this:

```
; 16-bit code (D=0)
    push bp
    mov bp, sp
    db 66h                ; operand size override — next instruction is 32-
bit
    call far [bp+4]       ; reads 6 bytes: selector (2) + 32-bit offset (4)
    mov sp, bp
    pop bp
    retf
```

A selector with D=1 is loaded → the processor switches to 32-bit mode for instructions at `lpFunction`. Return from the 32-bit function must be via `retf` (or `retf` with 66h prefix if returning to a 16-bit segment).

2.5 Algorithm of ''CallProc32W'' and ''CallProcEx32W''

Pseudo-code based on Finnegan's article and reverse engineering:

```
DWORD CallProcEx32W(DWORD nParams, DWORD fAddrConv, DWORD lpFunc, ...) {
    DWORD args[32];
    va_list valist;
    va_start(valist, lpFunc);
    for (int i = 0; i < (nParams & 0x1F); i++) {
        DWORD arg = va_arg(valist, DWORD);
        if (fAddrConv & (1 << i)) {
            // convert 16:16 to linear via GetVDMPointer32W
```

```
            args[i] = GetVDMPointer32W((LPVOID)(DWORD)arg, 1);
        } else {
            args[i] = arg;
        }
    }
    va_end(valist);

    // switch to 32-bit stack (inside NTVDM)
    // copy args to 32-bit stack
    // perform far call with 66h prefix to lpFunc
    // copy result from EAX to return DWORD
    // restore 16-bit stack
    return eax_result;
}
```

2.6 Memory management: ''GlobalFix'' / ''GlobalUnfix''

Microsoft documentation (Win32 SDK for NT) requires: if you pass a pointer to data in a **moveable** segment (allocated with `GlobalAlloc` and GMEM_MOVEABLE) to `CallProcEx32W`, you must fix that segment before the call and unfix it after.

```
HGLOBAL hMem = GlobalAlloc(GMEM_MOVEABLE, 1024);
LPSTR p = GlobalLock(hMem);
GlobalFix(hMem);   // prevent moving
// call CallProcEx32W with p
GlobalUnfix(hMem);
GlobalUnlock(hMem);
```

This is less critical on Windows NT due to virtual memory, but on Windows 95/98 (where Generic Thunk partially worked) it was mandatory.

2.7 Full code example (16-bit → 32-bit)

```
// 16-bit module (WOWTST16.EXE)
#include <windows.h>

HINSTANCE32 hK32;
FARPROC pfnGetTickCount;

void InitThunk() {
    hK32 = LoadLibraryEx32W("KERNEL32.DLL", NULL, 0);
    pfnGetTickCount = GetProcAddress32W(hK32, "GetTickCount");
}

DWORD GetTickCount32() {
    return CallProcEx32W(1, 0, (DWORD)pfnGetTickCount, 0);
}
```

```
void CallWithString(LPSTR str) {
    DWORD linear = GetVDMPointer32W(str, 1);
    // call 32-bit function that expects LPSTR
    CallProcEx32W(1, PARAM_01, (DWORD)pfnSomeFunc, linear);
}
```

2.8 Limitations and known issues

* **Supported only on Windows NT/2000/XP** (officially not supported on Windows 95/98/Me, though some versions might partially work). * Maximum number of parameters – **32** (due to DWORD bitmask). * Cannot pass a 16-bit callback pointer to a 32-bit function (e.g., `EnumWindows`). * `CallProc32W` uses Pascal calling convention, making it hard to call varargs functions (use `CallProcEx32W` instead). * Manual memory management with `GlobalFix` is required.

3. Universal Thunk (Win32s)

3.1 Win32s architecture and place of Universal Thunk

Win32s is an add-on component for Windows 3.1 that allows running 32-bit applications in a 16-bit Windows environment. It is implemented as a set of thunks that translate Win32 API calls to 16-bit counterparts. Universal Thunk is an extension that allows a **32-bit application** to call arbitrary **16-bit DLLs** (and, in a limited form, vice versa).

3.2 Four-component UT ecosystem

To use Universal Thunk, four components are needed:

32-bit application (EXE) – wants to use a 16-bit DLL. # **32-bit interface DLL** – contains code that calls `UTRegister`. # **16-bit interface DLL** – exports `UT16Init` and `UT16Proc`. # **Target 16-bit DLL** – contains the real logic (may be the same as #3).

The 32-bit interface DLL and the 16-bit interface DLL are linked via `UTRegister`. Win32s automatically loads the 16-bit DLL and calls its initialisation routine.

3.3 Universal Thunk API

All functions are exported from `KERNEL32.DLL` (in the Win32s environment). Prototypes in `w32sut.h` (from Win32 SDK).

```
BOOL UTRegister(
    HANDLE hModule,                 // handle of 32-bit DLL
(GetModuleHandle(NULL) for EXE)
    LPCSTR lpsz16BitDLL,            // name of 16-bit interface DLL (no
path)
    LPCSTR lpszInitFunc,            // name of init function (UT16Init)
```

```
    LPCSTR lpszStepdownFunc,        // name of dispatcher (UT16Proc)
    UT32PROC *ppfnStepdownThunk,    // [out] pointer to stepdown thunk
(32→16)
    FARPROC pfnCallback32,          // [in] 32-bit callback (NULL if not
needed)
    LPVOID lpvData                  // [in] data for UT16Init (translated to
16:16)
);
void UTUnRegister(HANDLE hModule);
LPVOID UTSelectorOffsetToLinear(LPVOID lp16);   // 16:16 → 0:32
LPVOID UTLinearToSelectorOffset(LPVOID lp32);   // 0:32 → 16:16
```

3.4 Initialisation protocol: ''UT16Init'' and ''UT16Proc''

The 16-bit interface DLL must export two functions (often with ordinals 2 and 3):

```
// UT16Init — called by Win32s after loading the 16-bit DLL
DWORD FAR PASCAL UT16Init(UT16CBPROC pfnCallback32, LPVOID lpvData) {
    // pfnCallback32 — pointer to stepup thunk (allows 16-bit code to call
32-bit)
    // lpvData — data from UTRegister (translated to 16:16)
    // save pfnCallback32 for future callbacks
    return TRUE; // return FALSE to abort registration
}

// UT16Proc — dispatcher called via stepdown thunk
DWORD FAR PASCAL UT16Proc(LPVOID lpvData, DWORD dwFunctionCode) {
    switch (dwFunctionCode) {
        case 1: return Some16BitFunction(lpvData);
        case 2: return AnotherFunction(lpvData, ...);
    }
    return 0;
}
```

Stepdown thunk – a pointer returned via `ppfnStepdownThunk`. 32-bit code can call it, passing data and a function code. Win32s translates the call into `UT16Proc`.

Stepup thunk – a pointer passed to `UT16Init`. 16-bit code can call it to execute a 32-bit callback function.

3.5 Translation lists (xlist)

For passing complex structures (with many pointers), the third parameter of the stepdown thunk can be a translation list. Convention: when calling the stepdown thunk, three parameters are passed: `lpvData`, `dwFunctionCode`, `LPVOID *xlist`. If `xlist` is not NULL, Win32s traverses the array (until NULL) and for each pointer (which is a pointer to a pointer) translates it from 0:32 to 16:16 before calling `UT16Proc`.

Example (from Oney's article):

```
LPVOID *xlist = malloc(4 * sizeof(LPVOID));
xlist[0] = &stepdownargs.format;  // pointer to string
xlist[1] = &stepdownargs.substargs;
xlist[2] = NULL;
DWORD result = (*stepdownThunk)(&stepdownargs, 0, xlist);
```

3.6 Memory management issues: 256 selectors, 32KB limit, ''GlobalAlloc''

256 selector problem: Win32s creates an alias selector for each 64KB block of virtual memory allocated via `VirtualAlloc`, so that 16-bit code can access it. The LDT can hold at most 256 selectors for this purpose. If the application allocates many small blocks (e.g., via new or `HeapAlloc`), selectors run out and `UTLinearToSelectorOffset` starts returning NULL.

Solution: use `GlobalAlloc` with GMEM_MOVEABLE. Global memory uses a different translation mechanism (via the GDI heap) that does not consume LDT selectors.

32KB limit: Win32s imposes a limit: the maximum size of a memory block that can be safely passed through UT is 32KB. This is because only one alias selector is created per block, and accessing beyond 32KB may cause a fault.

Additionally, before calling the stepdown thunk, if you pass a pointer to moveable memory (`GlobalAlloc` with GMEM_MOVEABLE), you must fix it with `GlobalFix`.

3.7 Real-world example: Tcl/Tk

In the Tcl 8.x source code for Windows, Universal Thunk was used to call 16-bit functions. Snippet:

```
// tclWin32s.c
HINSTANCE hKernel = LoadLibrary("KERNEL32.DLL");
UTREGISTER utRegister = (UTREGISTER)GetProcAddress(hKernel, "UTRegister");
UTUNREGISTER utUnregister = (UTUNREGISTER)GetProcAddress(hKernel,
"UTUnRegister");
if (utRegister) {
    utRegister(hInst, "TCL16.DLL", "UT16Init", "UT16Proc", &stepdown, NULL,
NULL);
}
// ...
utUnregister(hInst);
```

3.8 Limitations and undocumented features

* **Win32s only**: Universal Thunk does not work on Windows NT, 95, or 98. * **One UT per module**: Each 32-bit module can register only one Universal Thunk. * **Complexity**: Requires two interface DLLs. * **Unreliable**: Because of the selector limitation, crashes are common. * **No support**: Microsoft discontinued Win32s after Windows 95.

4. Flat Thunk (Windows 95/98/Me)

4.1 Windows 9x architecture and bidirectional calls

Windows 95/98 has two independent kernels: * **16-bit** (KRNL386.EXE, USER.EXE, GDI.EXE) – for legacy applications. * **32-bit** (KERNEL32.DLL, USER32.DLL, GDI32.DLL) – for new applications.

They communicate via built-in internal thunks. **Flat Thunk** is a mechanism that allows user code to create similar bidirectional bridges. The key difference from Generic/Universal Thunk: a single tool **Thunk.exe** generates a pair of DLLs (16 and 32) that automatically translate calls in both directions.

4.2 Quick Thunk and the "QT_Thunk" function

At the heart of Flat Thunk lies an **undocumented function** QT_Thunk, exported from KERNEL32.DLL. Its prototype (reconstructed by Matt Pietrek):

```
DWORD QT_Thunk(DWORD functionIndex, DWORD argCount, DWORD *args);
```

* `functionIndex` – index of the function in the 16-bit dispatcher table (generated by Thunk.exe). * `argCount` – number of arguments (up to 32). * `args` – array of DWORD values to be passed to the 16-bit code.

QT_Thunk performs: # Saves 32-bit context (registers, stack). # Switches to a 16-bit stack (SS:SP). # For each argument: if the argument is a pointer (determined by the .THK description), converts 0:32 → 16:16. # Calls the 16-bit code via a far call. # Converts the result from DX:AX to EAX. # Restores the 32-bit context. # Returns the result.

4.3 Algorithm of "QT_Thunk" (according to Matt Pietrek)

Pseudo-code from “Windows 95 System Programming Secrets” (1996):

```
DWORD QT_Thunk(DWORD func, DWORD nArgs, DWORD *pArgs) {
    // save FS, GS, EBP, ESI, EDI, DS, ES, SS
    // switch DS to 32-bit data selector
    // switch stack: esp -> temporary 16-bit stack
    for (i=0; i<nArgs; i++) {
        DWORD arg = pArgs[i];
        if (bit i set in thunk descriptor) {
            // convert 0:32 to 16:16 (allocate selector)
            push seg, off
        } else {
            push low16(arg), high16(arg) // actually push DWORD
        }
    }
    push func (index)
    call far [16-bit dispatcher]
    // result in DX:AX
```

```
    // restore context
    return (DWORD)AX | ((DWORD)DX << 16);
}
```

4.4 Thunk Compiler (THUNK.EXE): .THK file format

Thunk.exe (included in the Platform SDK for Windows 95) reads a text `.THK` file and generates an assembly `.ASM` file. Example:

```
// sample.thk
typedef struct tagPOINT {
    int x;
    int y;
} POINT;

BOOL WINAPI GetCursorPos(POINT FAR* lpPoint) = 16;  // function on the 16-bit side

param(lpPoint) = inout;   // pointer is translated and data copied both ways
```

Directives: * `= 16` – function resides in a 16-bit DLL (called from 32-bit). * `= 32` – function resides in a 32-bit DLL (called from 16-bit). * `param(parameter) = in | out | inout` – direction of data transfer (for pointer translation).

Thunk.exe generates a single `.ASM` file containing: - A function description table (for `QT_Thunk`). - 32-bit proxy functions (to be called from 32-bit code). - 16-bit proxy functions (to be called from 16-bit code).

4.5 Building a 16-bit and 32-bit DLL pair with Thunk.exe

Steps:

Create `mylib.thk` with function descriptions. # Run `thunk mylib.thk mylib.asm`. # Build the 32-bit DLL:

```
<code bash>
ml /c /DIS_32 mylib.asm
cl /c /GD mylib32.c
link mylib32.obj mylib32.obj /DLL /OUT:MYLIB32.DLL
</code>
```

Build the 16-bit DLL:

```
<code bash>
ml /c /DIS_16 mylib.asm
cl /c /AS /G2 mylib16.c
link mylib16.obj mylib16.obj /DLL /OUT:MYLIB16.DLL
</code>
```

The 32-bit application calls `MYLIB32.DLL`, which uses `QT_Thunk` to talk to `MYLIB16.DLL`.

4.6 Differences between Flat Thunk and Generic/Universal Thunk

Feature	Flat Thunk	Generic Thunk	Universal Thunk
Platform	Windows 95/98	Windows NT	Win32s
Direction	16⇄32 bidirectional	16→32	32→16
Code generation	Thunk.exe	manual via API	manual via UTRegister
Platform dependence	High	Medium	Very high
Callback support	Yes (via .THK)	No	Yes (via stepup thunk)

5. Thunklet (low-level WOW building block)

5.1 Definition and role

Thunklet is the smallest executable block of code (usually 16 bytes) that performs bitness switching and transfers control between 16-bit and 32-bit code. Thunklet is not a public API; it is an internal mechanism of the Windows NT kernel (WOW) and, to some extent, Windows 95. All high-level thunks (Generic, Flat) are built from thunklets.

5.2 ''_THUNKLET'' structure (16 bytes)

From the Wine project (`include/wine/thunk.h`):

```
typedef struct _THUNKLET {
    WORD    opcodes[4];      // 8 bytes of machine code
    DWORD   lpFunction;      // 4 bytes: 32-bit linear address of target function
    WORD    wRelayID;        // 2 bytes: unique identifier (for callbacks)
    WORD    wReserved;       // 2 bytes: alignment
} THUNKLET;
```

opcodes – machine code that when executed: - Saves registers. - Loads the 32-bit address from `lpFunction`. - Switches bitness (66h prefix) and performs a `call far`. - Restores registers and returns.

5.3 Client Thunklet and Server Thunklet

* **Client Thunklet** (16-bit → 32-bit call): Called from 16-bit code. It switches the processor to 32-bit mode and calls the function at `lpFunction`. After return, it switches back. * **Server Thunklet** (return / callback 32-bit → 16-bit): Used for callbacks. It stores the 16-bit return address and stack selector. When called, it switches to the 16-bit stack and passes control.

5.4 Hidden KERNEL Thunklet API (ordinals 560–568, 604–612, 619–622)

The 16-bit KRNL386.EXE (the version for Windows NT) contains additional exports intended for internal WOW use. They are undocumented but known from reverse engineering (Wine, IDA Pro).

Ordinal range	Presumed purpose
560–568	Thunklet management: AllocThunklet, FreeThunklet, GetThunklet, SetThunkletFunction
604–612	Address translation: LinearToSelector, SelectorToLinear, FixPointer
619–622	Client callbacks: CallClientThunk, ReplyClientThunk

Example hypothetical call:

```
// Get Thunklet by ID (undocumented, ordinal 560)
THUNKLET FAR* GetThunklet(WORD wRelayID, WORD wType);

// Bind Thunklet to a function (ordinal 562)
void SetThunkletFunction(THUNKLET FAR* pThunk, DWORD lpFunction);
```

5.5 Example machine code of a Thunklet (from Windows NT)

Disassembled Thunklet from KRNL386.EXE (16-bit segment):

```
; Client Thunklet (call 32-bit function)
    push    bp
    mov     bp, sp
    db      66h                 ; operand size override
    call    far [bp+4]          ; call address stored in lpFunction
    mov     sp, bp
    pop     bp
    retf
```

Here [bp+4] is the location on the stack where the 32-bit address (from lpFunction) is placed before the call. The 66h prefix forces the processor to interpret the call far as 32-bit (reads 6 bytes: selector+offset).

5.6 Implementation in Wine

Wine, emulating the Win32 API on Unix-like systems, implements thunklets to support 16-bit applications (NTVDM). In dlls/wow32/thunk.c:

```
THUNKLET *THUNK_Alloc(void) {
    THUNKLET *thunk = VirtualAlloc(NULL, sizeof(THUNKLET), MEM_COMMIT,
PAGE_READWRITE);
    // fill opcodes with default code (push bp, mov bp, sp, db 66h, call
...)
    memcpy(thunk->opcodes, defaultThunkCode, 8);
```

```
    thunk->wRelayID = 0;
    thunk->lpFunction = 0;
    return thunk;
}

void THUNK_SetFunction(THUNKLET *thunk, DWORD func) {
    thunk->lpFunction = func;
}
```

Wine also contains `elf_is_in_thunk_area`, used by debuggers to determine whether an address belongs to the thunklet area.

Comparison table of all mechanisms

Mechanism	OS	Direction	Thunk size	API / tool	Bitness switching	Pointer translation	Callback 16→32	Memory management
Instance Thunk	Win16 (3.x)	16→16 callback	8–16 bytes	`MakeProcInstance`	not needed (same bitness)	not needed (DS from AX)	N/A	not required
Generic Thunk	WinNT 3.1–4.0	16→32	built into `CallProc32W`	`LoadLibraryEx32W` etc.	66h prefix + far call	`GetVDMPointer32W`	no	`GlobalFix`
Universal Thunk	Win32s	32→16 (and back)	external (stepdown thunk)	`UTRegister`, `UTUnRegister`	via Win32s kernel	`UTSelectorOffsetToLinear`	yes (stepup thunk)	`GlobalFix`, 256 selectors, 32KB limit
Flat Thunk	Win95/98/Me	16⇄32 bidirectional	generated by THUNK.EXE	`QT_Thunk`, Thunk.exe	`QT_Thunk`	built into `QT_Thunk`	yes (via .THK)	`GlobalFix`, `GlobalAlloc`
Thunklet	internal WOW (NT/95)	low-level block	16 bytes	hidden API (KRNL386 ordinals)	66h prefix + jmp	via separate functions	via RelayID	not required

Conclusion

All five thunking mechanisms were developed during the transition from 16-bit Windows to 32-bit Windows. Each solved a specific problem on a specific platform:

* **Instance Thunk** – a legacy of Win16, solving the binding of callback functions to instance data. * **Generic Thunk** – an elegant but platform-specific solution for Windows NT, allowing 16-bit applications to use 32-bit DLLs. * **Universal Thunk** – a complex but only possibility for 32-bit applications on Win32s to call 16-bit code. * **Flat Thunk** – a powerful, bidirectional mechanism for Windows 95/98, using `QT_Thunk` and the Thunk Compiler. * **Thunklet** – the low-level “atom” from which all other thunks are built.

Understanding these mechanisms is essential for maintaining legacy systems, reverse engineering, and analysing historical code. Modern versions of Windows (starting with 2000) do not support any of these thunks (except emulating `MakeProcInstance` as a stub).

References

* Finnegan, J. “Test Drive Win32 from 16-bit Code Using the Windows NT WOW Layer and Generic Thunk”. Microsoft Systems Journal, June 1994. (PDF: `24.pdf`) * Oney, W. “Mix 16-bit and 32-bit Code in Your Applications with the Win32s Universal Thunk”. Microsoft Systems Journal, November 1993.

(PDF: `mix16.pdf`) * Pietrek, M. “Windows 95 System Programming Secrets”. IDG Books, 1996. (Chapter on `QT_Thunk`) * Petzold, C. “Programming Windows 3.1”. Microsoft Press, 1992. (Chapters on `MakeProcInstance`) * Wine source code: `dlls/wow32/thunk.c`, `include/wine/thunk.h` * Microsoft Win32 SDK for Windows NT 3.5, files: `WOWNT16.H`, `WOWNT32.H` * Microsoft Knowledge Base article Q104009: “Generic Thunk Interface in Windows NT”