Peeking inside the implementation of AnsiUpper and AnsiLower in Windows 1.0

devblogs.microsoft.com/oldnewthing/20200803-00

August 3, 2020

Raymond Chen

Windows 1.0 had functions called `AnsiUpper` and `AnsiLower`. You passed these functions a pointer to a string, and it converted the string in place to uppercase or lowercase, respectively. If the segment portion of the pointer is zero, then the offset is treated as a character code, and it returned the uppercase version of that character code in the low byte of the return value.

The single-character version could anachronistically be wrapped like this:[1]

```
inline char AnsiUpperChar(char c)
{
 return reinterpret_cast<char>(
    AnsiUpper(reinterpret_cast<LPSTR>(
      static_cast<unsigned char>(c))));
}
```

This is an anachronism because in 1983, there was no `reinterpret_cast`, no `static_cast`, no inline functions, and no C++.

It was more likely to be a macro.

```
#define AnsiUpperChar(c) ((char)AnsiUpper((LPSTR)(unsigned char)(c)))
```

The implementations of these functions is entirely in assembly language.

```
; Entry: pointer on stack
; Exit:  If single character, AL = converted character
;        If string, DX:AX = original pointer

AnsiUpper proc far
        mov bx, sp          ; custom stack frame
        push di             ; save registers
        push si
        les di, ss:[bx+4]   ; es:di -> string
        mov cx, es          ; cx:ax -> string
        mov ax, di
        call UpperChar      ; uppercase the character in AL
        jcxz aup90          ; Exit if CX = 0
        call UpperString    ; uppercase the string in ES:DI
        mov dx, es          ; return the original pointer
        mov ax, ss:[bx+4]
aup90:  pop si
        pop di
        ret 4
AnsiUpper endp

; Entry: AL = character
; Exit:  AL = uppercase version of character
; Modifies: No other registers

UpperChar proc near
        cmp al, 0x61        ; Q: Less than 'a'?
        jb uch90            ; Y: Nothing to do
        cmp al, 0x7a        ; Q: Less than 'z'?
        jbe uch80           ; Y: Convert to uppercase
        cmp al, 0xe0        ; Q: Less than 'à'?
        jb uch90            ; Y: Nothing to do
        cmp al, 0xfe        ; Q: More than 'þ'?
        ja uch90            ; Y: Nothing to do
uch80:  sub al, 0x20        ; Convert lowercase to uppercase
uch90:  ret
UpperChar endp

; Entry: ES:DI -> string to convert to uppercase
; Exit: String has been converted to uppercase in place
; Modifies: SI, DI, AL

UpperString proc near
        cld                 ; Ensure we walk forward
        mov si, di          ; ES:SI and ES:DI both -> string
ust10:  lodsb es:[si]       ; Load character and advance SI
        call UpperChar      ; Convert to uppercase
        stosb               ; Save result and advance DI
        or al, al           ; Q: End of string?
        jnz ust10           ; N: Keep converting
        ret
UpperString endp
```

The `AnsiLower` function is entirely analogous, so I won't bother writing it out.

The `AnsiUpper` function doesn't use the usual `BP` stack frame. To save code space, it uses `BX` as the stack frame pointer. That way, it doesn't need to do all the usual frame setup and teardown stuff. This code does not call out to other code segments, so we won't trigger any segment-not-present thunks that would require stack patching, so the lack of a proper `BP` frame is not going to cause a problem.

The structure of the `AnsiUpper` function is rather odd. It first assumes that you're calling it with a single character and converts the offset from lowercase to uppercase. Only after the conversion does it check whether you actually called it that way. If so, then it jumps to the exit with the converted character. Otherwise, it throws away all the work it did and starts over by converting the pointed-to string.

Why does it structure the code this way? Because it saves an instruction. Instead of

```
    if condition goto branch2
    do_branch1
    goto end
branch2:
    do_branch2
end:
```

you speculatively front-load one of the branches and discard it if it turns out to be the wrong branch.

```
    do_branch2
    if condition goto end
    do_branch1
end:
```

This removes the `goto end` from the instruction stream, saving two bytes.

Of course, this trick requires that `do_branch2` has no side effects, or at least that the side effects can be rolled back if the speculation turns out to have been unwarranted.

The `UpperChar` function has a custom register-based calling convention. This is common in hand-written assembly language, allowing you to tailor the calling convention to the usage pattern.

You may have noticed that the `UpperChar` function doesn't consult any code page tables to figure out which characters are uppercase and which are lowercase. It just hard-codes the special knowledge of code page 1252, which was the ANSI code page that Windows 1.0 used.

In the layout of code page 1252, the letters are in two blocks: One from A to Z, and another from À to Þ. Furthermore, the uppercase and lowercase versions are exactly 32 slots apart, so adding 32 gets you from uppercase to lowercase, and subtracting 32 gets you from lowercase

to uppercase.

Okay, back to `AnsiUpper`. If it turns out that we have a string, then the work is done by the `UpperString` function. This function takes advantage of the special `LODSB` and `STOSB` instructions to load a single byte from the string and to write a single byte to the string. These are single-byte instructions that replace two larger instructions (load a byte and increment the index register), so they are handy when trying to squeeze every code byte out of your program.

You may have spotted some quirks in this conversion code.

The `CharUpper` function treats U+00D7 × as the uppercase version of U+00F7 ÷. If you ask for the lowercase version of the multiplication symbol, you get the division symbol, and conversely when converting from lowercase to uppercase.

Another quirk is that the code doesn't try to capitalize ß to SS. It just leaves it as ß. There is no uppercase ẞ in code page 1252.

Believe it or not, there was a point to this exercise beyond just digging up ancient code designed under very different constraints and marveling how it worked. We'll put this function into context next time.

[1] You might be tempted to use this:

```
inline char AnsiUpperChar(char c)
{
 return reinterpret_cast<char>(
    AnsiUpper(reinterpret_cast<LPSTR>(c)));
}
```

but that doesn't work because `char` is probably a signed type, so the `char` will be sign-extended, which means that a character in the `0x80` to `0xFF` range will produce a pointer of the form `0xFFFF:0xFFxx`. Since this does not have zero in the high word, it will be treated as a pointer and corrupt random memory.

Raymond Chen

Follow