Why are there trivial functions like CopyRect and EqualRect?

devblogs.microsoft.com/oldnewthing/20200224-00

February 24, 2020

Raymond Chen

If you dig into the bag of tricks inside `user32`, you'll see some seemingly-trivial functions like `CopyRect` and `EqualRect`. Why do we even need functions for things that could be done with the `=` and `==` operators?

Because those operators generate a lot of code.

Copying a rectangle would go like this:

```
c4 5e f0        les  bx, [bp-10]    ; es:bx -> source rect
26 8b 07        mov  ax, es:[bx]    ; ax = source.left
c4 5e ec        les  bx, [bp-14]    ; es:bx -> destination rect
26 89 07        mov  es:[bx], ax    ; dest.left = ax

c4 5e f0        les  bx, [bp-10]    ; es:bx -> source rect
26 8b 47 02     mov  ax, es:[bx+2]  ; ax = source.top
c4 5e ec        les  bx, [bp-14]    ; es:bx -> destination rect
26 89 47 02     mov  es:[bx+2], ax  ; dest.top = ax

c4 5e f0        les  bx, [bp-10]    ; es:bx -> source rect
26 8b 47 04     mov  ax, es:[bx+4]  ; ax = source.right
c4 5e ec        les  bx, [bp-14]    ; es:bx -> destination rect
26 89 47 04     mov  es:[bx+4], ax  ; dest.right = ax

c4 5e f0        les  bx, [bp-10]    ; es:bx -> source rect
26 8b 47 06     mov  ax, es:[bx+6]  ; ax = source.bottom
c4 5e ec        les  bx, [bp-14]    ; es:bx -> destination rect
26 89 47 06     mov  es:[bx+6], ax  ; dest.bottom = ax
```

This takes 54 bytes of code. It's rather inefficient because the 8086 processor could indirect only through the `bx`, `bp`, `si`, and `di` registers. The `bp` register was reserved for use as the frame pointer, so that was off the table. The `si` and `di` registers were used as register variables, so they are busy holding something important. That leaves `bx` as the only register that can be used to dereference pointers.

Since this is a 16:16 pointer, we also need a segment register, and the 8086 has only four segment registers: `cs` (code segment), `ds` (data segment), `ss` (stack segment), `es` (extra segment). Three of them have dedicated purposes, so the only one left is `es`. Even if we could borrow `si` or `di` temporarily, we would still be bottlenecked on `es`.

If we move `CopyRect` to a function, then we can save a bunch of code:

```
c4 5e f0        les  bx, [bp-10]    ; es:bx -> source rect
53              push bx
06              push es
c4 5e ec        les  bx, [bp-14]    ; es:bx -> destination rect
53              push bx
06              push es
9a xx xx xx xx  call CopyRect
```

Only 15 bytes. Less than a third the size.

This was the era in which developers counted bytes, and any trick to save a few bytes was worth considering, especially since you had “only” 256KB of memory.[1]

And since copying and comparing rectangles were common operations, factoring the code into a function saved a lot of bytes.

Of course, nowadays, it’s not a lot of code to copy a rectangle manually: An entire rectangle fits into a single 128-bit register.

```
    mov    eax, [sourcerect]
    movups xmm0, [eax]
    mov    eax, [destrect]
    movups [eax], xmm0
```

Bonus code golf: We could have squeezed out a few instructions by moving two integers at a time. This requires that the two rectangles be non-overlapping in memory (to avoid data aliasing), but that’s probably a safe assumption because the original code didn’t work anyway in that case.

```
int v[5];
*(RECT*)&v[0] = *(RECT*)&v[1]; // bad idea
```

Switching to moving two integers at a time doesn’t break anything that wasn’t already broken, so let’s do it:

```
c4 5e f0        les  bx, [bp-10]    ; es:bx -> source rect
26 8b 07        mov  ax, es:[bx]    ; ax = source.left
26 8b 57 02     mov  dx, es:[bx+2]  ; dx = source.top
c4 5e ec        les  bx, [bp-14]    ; es:bx -> destination rect
26 89 07        mov  es:[bx], ax    ; dest.left = ax
26 89 57 02     mov  es:[bx+2], dx  ; dest.top = dx

c4 5e f0        les  bx, [bp-10]    ; es:bx -> source rect
26 8b 47 04     mov  ax, es:[bx+4]  ; ax = source.right
26 8b 57 06     mov  dx, es:[bx+6]  ; dx = source.bottom
c4 5e ec        les  bx, [bp-14]    ; es:bx -> destination rect
26 89 47 04     mov  es:[bx+4], ax  ; dest.right = ax
26 89 57 06     mov  es:[bx+6], dx  ; dest.bottom = dx
```

That dropped us down to 42 bytes. It helps, but it's still a lot of code.

If we're willing to spill one of our other register variables, say, `si`, then we can squeeze it even further.

```
c4 5e f0        les  bx, [bp-10]    ; es:bx -> source rect
26 8b 07        mov  ax, es:[bx]    ; ax = source.left
26 8b 57 02     mov  dx, es:[bx+2]  ; dx = source.top
26 8b 4f 04     mov  cx, es:[bx+4]  ; cx = source.right
26 8b 77 06     mov  si, es:[bx+6]  ; si = source.bottom
c4 5e ec        les  bx, [bp-14]    ; es:bx -> destination rect
26 89 07        mov  es:[bx], ax    ; dest.left = ax
26 89 57 02     mov  es:[bx+2], dx  ; dest.top = dx
26 89 4f 04     mov  es:[bx+4], cx  ; dest.right = cx
26 89 77 06     mov  es:[bx+6], si  ; dest.bottom = si
```

Only 36 bytes. Getting better. But still twice as big as calling `CopyRect`, and it cost us a register.

Another trick: Copy the rectangle through the stack.

```
c4 5e f0        les  bx, [bp-10]    ; es:bx -> source rect
26 ff 37        push es:[bx]        ; push source.left
26 ff 77 02     push es:[bx+2]      ; push source.top
26 ff 77 04     push es:[bx+4]      ; push source.right
26 8b 77 06     push es:[bx+6]      ; push source.bottom
c4 5e ec        les  bx, [bp-14]    ; es:bx -> destination rect
26 8f 47 06     pop  es:[bx+6]      ; pop dest.bottom
26 8f 47 04     pop  es:[bx+4]      ; pop dest.right
26 8f 47 02     pop  es:[bx+2]      ; pop dest.top
26 8f 47        pop  es:[bx]        ; pop dest.left
```

Hm, same code size as using registers.

Okay, how about borrowing the `ds` register as well the `si` and `di` registers?

```
1e              push ds
c5 7e ec        lds  di, [bp-14]
c4 76 f0        les  si, [bp-10]
fc              cld
a5              movsw
a5              movsw
a5              movsw
a5              movsw
1f              pop  ds
```

Thirteen bytes, yay, though it did cost us register spills that are not immediately visible.

This version is a tightrope walk because any operation that yields the processor risks discarding the former `ds` segment, which will cause problems because we will restore it to an invalid value and corrupt memory!

[1] The word “only” in in quotation marks because 256KB seems like a tiny amount of memory today, but at the time, that was the maximum amount of memory you could get for an IBM PC XT! At least not without resorting to expansion cards.

Raymond Chen

Follow