What's with the weird wReserved value at the start of the DECIMAL structure?

devblogs.microsoft.com/oldnewthing/20250516-00/?p=111185

May 16, 2025

The `DECIMAL` structure looks like this:

```
typedef struct tagDEC {
  USHORT wReserved;
  union {
    struct {
      BYTE scale;
      BYTE sign;
    };
    USHORT signscale;
  };
  ULONG  Hi32;
  union {
    struct {
      ULONG Lo32;
      ULONG Mid32;
    };
    ULONGLONG Lo64;
  };
} DECIMAL;
```

What is the deal with that `wReserved` at the front?

That reserved field comes into play when the `DECIMAL` is placed inside a `VARIANT`.

Let's start with this simple version of `VARIANT`:

```
typedef struct tagVARIANT {
  uint16_t vt;
  union {
    uint64_t    llVal;
    uint32_t    lVal;
    uint8_t     bVal;
    int16_t     iVal;
    ⟦ a whole bunch of other things ⟧
    ⟦ the largest is 8 bytes        ⟧
    ⟦ and requires 8-byte alignment ⟧
  };
} VARIANT;
```

This simple version is a discriminated union with a large number of possible types. The largest such type is 8 bytes in size and has an alignment of 8 bytes. This means that there are 6 bytes of padding between the discriminant `vt` and the union.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
vt		(padding)						payload							

Let's add the padding explicitly.

```
typedef struct tagVARIANT {
  uint16_t vt;
  uint16_t pad1;
  uint16_t pad2;
  uint16_t pad3;
  union {
    uint64_t    llVal;
    uint32_t    lVal;
    uint8_t     bVal;
    int16_t     iVal;
    [[ a whole bunch of other things ]]
    [[ the largest is 8 bytes        ]]
    [[ and requires 8-byte alignment ]]
  };
} VARIANT;
```

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
vt		pad1		pad2		pad3		payload							

Now, the `DECIMAL` type wants to hold a scaled decimal value, so we'll need at least a byte for the scale, a bit for the sign, and a lot of bits for the mantissa. If we had to squeeze the `DECIMAL` inside the payload field, there would be only 64 bits available. But wait, there are these extra 48 bits of padding. Maybe we can steal them to hold the scale, sign, and more mantissa bits.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
vt		(I CAN HAZ MOAR PAYLOAD?)						payload							

Great, we found an additional place to stash 48 bits of information.

Our goal is to have the `VARIANT` laid out like this:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
vt		pad1		pad2		pad3		payload								for non-`DECIMAL`
		scale	sign	mantissa_high				mantissa_low								for `DECIMAL`

So let's declare our `DECIMAL` structure so that it can nestle right in after the `vt`:

```
typedef struct tagDECIMAL
{
  uint8_t scale;
  uint8_t sign;
  uint32_t mantissa_high;
  uint64_t mantissa_low;
} DECIMAL;

typedef struct tagVARIANT
{
  uint16_t vt;
  union {
    DECIMAL decVal;
    struct {
      uint16_t pad1;
      uint16_t pad2;
      uint16_t pad3;
      union {
        uint64_t    llVal;
        uint32_t    lVal;
        uint8_t     bVal;
        int16_t     iVal;
        [[ a whole bunch of other things ]]
        [[ the largest is 8 bytes        ]]
        [[ and requires 8-byte alignment ]]
      };
    };
  };
};
```

Oh noes! The result is

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	
vt		(padding)						pad1		pad2		pad3		(padding)		payload								for non-DECIMAL
		(padding)						scale	sign	(padding)		mantissa_high				mantissa_low								for DECIMAL

Our attempt to squeeze out the padding just resulted in even more padding!

The problem is that the DECIMAL structure requires padding in order to align the two mantissa fields. There is no way to tell the compiler to lay out a structure on the assumption that it always appears as part of a larger structure which misaligns its start address. Mind you, even if there were a way to do it, you wouldn't want to, because code is allowed to declare a standalone DECIMAL structure that is not part of a VARIANT, and now your promise is broken.

The unnamed structure we introduced into our VARIANT has the same problem as DECIMAL, so it too gets alignment padding so that the unnamed structure starts at a multiple of 8.

To get the layout we want, we must move the initial vt *inside the* *DECIMAL*.

```
typedef struct tagDECIMAL
{
  uint16_t reserved;
  uint8_t scale;
  uint8_t sign;
  uint32_t mantissa_high;
  uint64_t mantissa_low;
} DECIMAL;

typedef struct tagVARIANT
{
  union {
    struct {
      uint16_t vt;
      uint16_t pad1;
      uint16_t pad2;
      uint16_t pad3;
      union {
        uint64_t    llVal;
        uint32_t    lVal;
        uint8_t     bVal;
        int16_t     iVal;
        [[ a whole bunch of other things ]]
        [[ the largest is 8 bytes        ]]
        [[ and requires 8-byte alignment ]]
      };
    };
    DECIMAL decVal;
  };
};
```

The resulting layout is now

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
vt		(padding)						payload								for non-DECIMAL
reserved		scale	sign	mantissa_high				mantissa_low								for DECIMAL

When the DECIMAL is not part of a VARIANT the reserved field is unused. But if it's part of a VARIANT, it's living a double life as a vt, and the value will always be VT_DECIMAL.

If a VARIANT represents a DECIMAL, you have the weird case that the vt comes from the VARIANT, but the other fields come from the DECIMAL. Back in the days when the VARIANT structure was defined, this sort of data punning was commonplace, and nobody would have batted an eye.

Fortunately, even though compiler-writers might cast side-eye at this sort of thing, the trick is legal in this case because the vt and wReserved fields are both defined as unsigned short and therefore satisfy the "common initial sequence" rule. This rule permits accessing the part of a non-active union member that shares a common initial sequence with the active union member. The basic idea is that you look for the longest stretch of initial elements of the two union members that agree in type, and those members can be read from one union member even if the other union member is the active member. (There is of course lots of fine print, but that's the basic idea.)

Bonus reading: Type Punning, Strict Aliasing, and Optimization by John Regehr.