# What possible use are those extra bits in kernel handles? Part 2: Overcoming limited expressiveness

devblogs.microsoft.com/oldnewthing/20080828-00

August 28, 2008

Raymond Chen

Last time, we saw how those extra bits can be used to develop safe sentinel values. That is a special case of a more general problem: How do you pack 33 bits of information into a 32-bit value? Whereas last time, we weren't forced into the use of a sentinel value because we could develop a (cumbersome) helper class and switch people over to the helper class (or to pass two parameters to every function that used to take one), there are places where you are forced to try to squeeze 33 bits of information into a 32-bit value, and the helper class simply isn't going to work. (I'm going to assume 32-bit Windows for concreteness, but the same considerations apply to 64-bit Windows. Just make the appropriate changes.)

Suppose you have a window message that does some work and returns a `HANDLE`, but it can fail, and when it does, you want to return an error code. In other words, you want to return either `(TRUE, HANDLE)` or `(FALSE, HRESULT)`. But that's 33 bits of information, and you can return only 32 bits. How can you provide 33 bits of information with only 32 bits?

Well, it turns out that you don't actually have 33 bits of information to return. Since handle values are multiples of four, the bottom two bits are always zero and therefore convey no information. A kernel handle is really only 30 bits. Similarly, a COM error code in the form of an `HRESULT` always has the top bit set—after all if the top bit were clear, it would be a success code! Therefore, there are only 31 bits of information in an `HRESULT` error code.

Okay, so it turns out that `(TRUE, HANDLE)` is only 1 + 30 = 31 bits of information, and `(FALSE, HRESULT)` is only 1 + 31 = 32 bits of information. We can fit them inside a single 32-bit value after all!

```
LRESULT PackHandleIntoLresult(HANDLE Handle)
{
  LRESULT Lresult = (LRESULT)Handle;


  // if this assertion fires, then somebody tried to
  // pack an invalid handle!
  assert((Lresult & 1) == 0);


  return Lresult;
}


LRESULT PackErrorHresultIntoLresult(HRESULT Hresult)
{
  // if this assertion fires, then somebody tried to
  // pack a success code!
  assert(FAILED(Hresult));


  return ((DWORD)Hresult << 1) | 1;
}
```

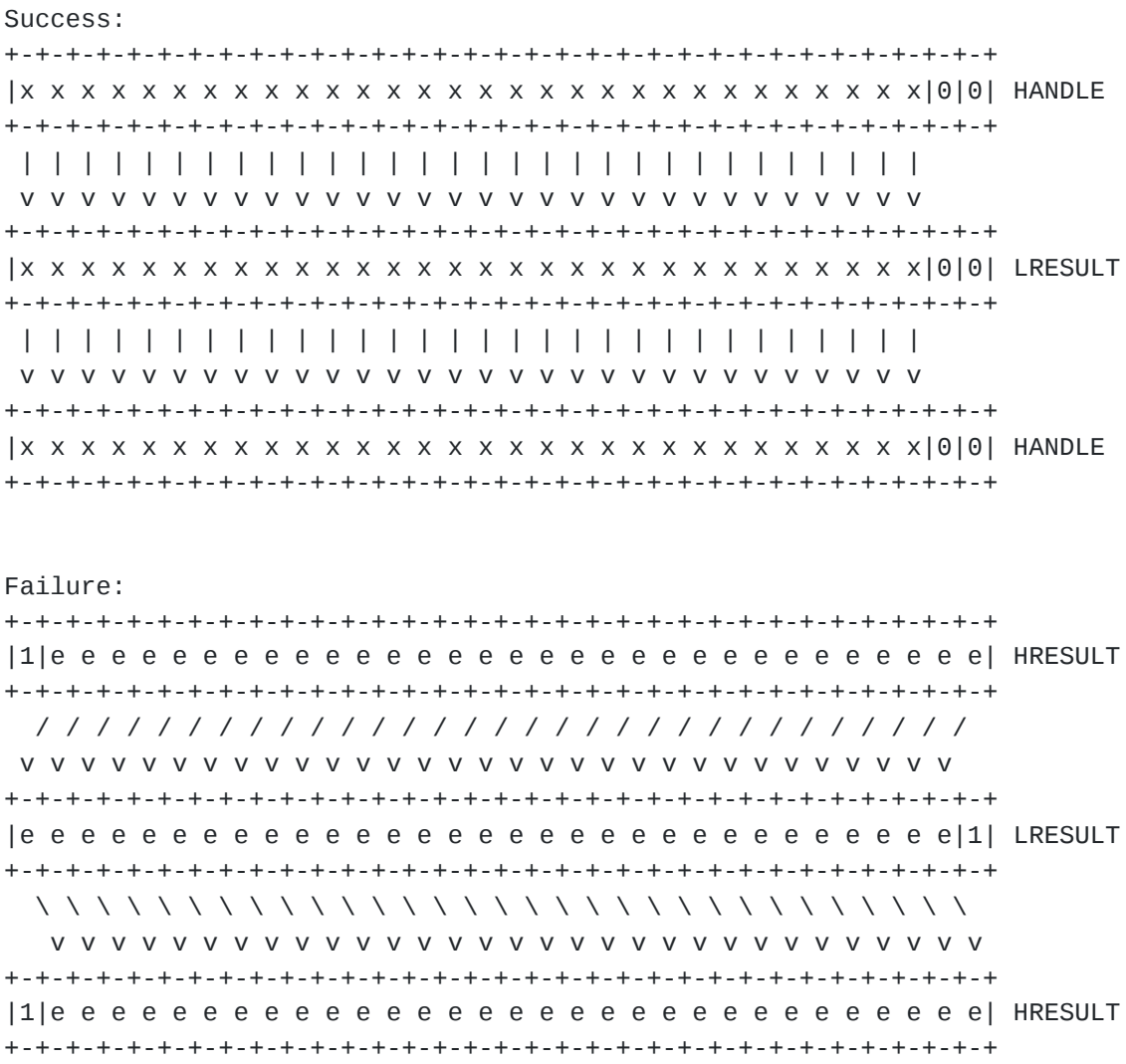
The bottom bit is our boolean that tells us whether we have a success or failure. If the bit is clear, then the operation succeeded and the entire value is the handle, relying on the fact that valid handles always have the bottom two bits clear. On the other hand, if the bottom bit is set, then we have an error code, and the remaining 31 bits give us the significant bits of the `HRESULT`. Unpacking the values would then go like this:

```
BOOL IsLresultError(LRESULT Lresult)
{
  return Lresult & 1;
}


HANDLE UnpackLresultToHandle(LRESULT Lresult)
{
  assert(!IsLresultError(Lresult));
  return (HANDLE)Lresult;
}


HRESULT UnpackLresultToHresult(LRESULT Lresult)
{
  assert(IsLresultError(Lresult));
  return (HRESULT)(0x80000000 | ((DWORD)Lresult >> 1));
}
```

In pictures (since people like pictures):




That bottom bit tells us whether the upper 31 bits are the meaningful bits from a `HANDLE` or the meaningful bits from an `HRESULT`. Once we know which case we are in, we can take those upper bits and put them back into meaningful parts of the source data.

If you want to put this trick on a more formal footing, you could express the multiplexing in the form of a discriminant in a union:

```
// Type-specific value for HANDLE is the upper 31 bits
LRESULT TypeSpecificValueFromHandle(HANDLE Handle)
{
  LRESULT Lresult = (LRESULT)Handle;


  // if this assertion fires, then somebody tried to
  // pack an invalid handle!
  assert((Lresult &1) == 0);


  // discard the bottom bit since we know it is zero
  return Lresult >> 1;
}


HANDLE HandleFromTypeSpecificValue(LRESULT Lresult)
{
  // regenerate the bottom bit which we know is zero
  return (HANDLE)(Lresult << 1);
}


// Type-specific value for HRESULT is the lower 31 bits
LRESULT TypeSpecificValueFromHresult(HRESULT Hresult)
{
  // if this assertion fires, then somebody tried to
  // pack a success code!
  assert(FAILED(Hresult));


  // discard the top bit since we know it is 1
  return (DWORD)Hresult & 0x7FFFFFFF;
}


HRESULT HresultFromTypeSpecificValue(LRESULT Lresult)
{
  // regenerate the top bit which we know is 1
  return (HRESULT)(Lresult | 0x80000000);
}


// Oh boy, let’s pack and unpack these puppies
#define TYPE_HANDLE  0
#define TYPE_HRESULT 1


typedef struct PACKEDLRESULT {
 int Type:1;
 LRESULT TypeSpecificValue:sizeof(LRESULT)*8-1;
} PACKEDLRESULT;
```

```
typedef union PACKEDLRESULTHELPER {
 PACKEDLRESULT Structure;
 LRESULT Lresult;
} PACKEDLRESULTHELPER;


LRESULT PackLresult(int Type, LRESULT TypeSpecificValue)
{
  PACKEDLRESULTHELPER Helper;
  Helper.Structure.Type = Type;
  Helper.Structure.TypeSpecificValue = TypeSpecificValue;
  return Helper.Lresult;
}


int GetPackedLresultType(LRESULT Lresult)
{
  PACKEDLRESULTHELPER Helper;
  Helper.Lresult = Lresult;
  return Helper.Structure.Type;
}


HANDLE GetHandleFromPackedLresult(LRESULT Lresult)
{
  PACKEDLRESULTHELPER Helper;
  Helper.Lresult = Lresult;
  return HandleFromTypeSpecificValue(Helper.Structure.TypeSpecificValue);
}


HRESULT GetHresultFromPackedLresult(LRESULT Lresult)
{
  PACKEDLRESULTHELPER Helper;
  Helper.Lresult = Lresult;
  return HresultFromTypeSpecificValue(Helper.Structure.TypeSpecificValue);
}
```

This more explicit form may be more helpful if you have more than just two types to discriminate among, but in our case, the extra typing probably just confuses the matter more than clearing it up.

This type of trick is actually quite common. For example, the `LresultFromObject` function uses a variation of this technique in order to pack a marshallable object *and* a COM error code into a single 32-bit value. It's also common in lisp systems, where it is known by the name *tagged pointers*.

Raymond Chen

**Follow**