How are BitBlt raster opcodes calculated?

devblogs.microsoft.com/oldnewthing/20180528-00

May 28, 2018

Raymond Chen

Commenter R P asks what the low-order 16 bits of the `BitBlt` raster opcodes mean.

The documentation explains that the high-order 16 bits of the raster opcode are the zero-extended 8-bit value that represents the result of the raster operation given the 8 combinations of three binary inputs (pattern, source, and destination). This is the easy part to understand.

The documentation also says that the low-order 16 bits are an operation code, but gives no information as to how the operation code is determined.

Okay, let's dig through the history.

Initially, the `BitBlt` raster opcodes were only 16-bit values, consisting of the operation codes. The higher-order 16 bits were added later because it turns out that people preferred table lookups to parsing operation codes.

Okay, that's enough beating around the bush. How do I parse the operation codes?

The operation code is interpreted as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op5		op4		op3		op2		op1		op6	template			bias	

Operations 1 through 5 select one of the following logical operations:

Value	Operation	Abbreviation
0	not	n
1	exclusive or	x
2	or	o

3	and	a

Operation 6 encodes an optional final *not* operation.

Value	Operation	Abbreviation
0	no operation	
1	not	n

The operations imply the number of input parameters required. Each binary operation pops two arguments off the stack and pushes the result back onto the stack, for a net reduction of one. The unary *not* operation pops one argument off the stack and pushes the result back onto the stack, for no net change. And when we're finished, we want one final result on the stack. Therefore, the number of items that need to be placed onto the stack is one more than the number of binary operations.

The template and bias tell you what those parameters are. First, the template selects one of the following sequences of elements.

Value	Template
0	SPDD
1	SPD
2	SDP
3	(not used)
4	(not used)
5	SSP*DS
6	SSP*PDS
7	SSD*PDS

Two of the templates are not currently used and are reserved for future expansion.

The bias specifies how many initial template elements to ignore. After that, take the template elements until you have consumed one more than the number of binary operations. Also take the asterisk, if you encounter one. And if you run out of template elements, then start over from the beginning.

Okay, now we put all the pieces together.

- For each template element:
    - If it is a letter, then push that input onto the stack.
    - If it is an asterisk, then perform the next operation.
- After you have used up all the template elements, perform all of the remaining operations.
- When you're done, there should be one value left on the stack. That is the result of the `BitBlt` operation.

Consider the raster opcode `0x00010289`. The operation index is 1, which decodes as follows:

P	1	1	1	1	0	0	0	0
S	1	1	0	0	1	1	0	0
D	1	0	1	0	1	0	1	0
Result	0	0	0	0	0	0	0	1

The operation code is `0x0289`, which decodes as follows:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
op5		op4		op3		op2		op1		op6	template			bias	
0	0	0	0	0	0	1	0	1	0	0	0	1	0	0	1
0		0		0		2		2		0	2			1	
n		n		n		o		o			SDP			1	

There are two binary operation codes (the two *or* operations), so we will need three parameters.

- The bias tells us to skip the first template element, so we skip the S.
- The next element in the template is a D, so we push the destination.
- The next element in the template is a P, so we push the pattern.
- We have run out of template elements, so we wrap around and see that we have an S, so we push the source.
- Now to use up the remaining operations, in order:
- Operation 1 tells us to pop the top two values from the stack, *or* them together, and push the result back onto the stack.

- Operation 2 tells us to pop the top two values from the stack, *or* them together, and push the result back onto the stack.
- Operation 3 tells us to pop the top value from the stack, *bitwise-not* it, and push the result back onto the stack.
- Operation 4 tells us to pop the top value from the stack, *bitwise-not* it, and push the result back onto the stack.
- Operation 5 tells us to pop the top value from the stack, *bitwise-not* it, and push the result back onto the stack.
- Operation 6 tells us to do nothing.
- The final value on the stack is the result of the `BitBlt` operation.

In RPN, this encodes compactly as `DPSoonnn`

It looks like a lot of work, but that's because I spelled it out in painstaking detail. A shorter version would be

- The template says SDP, and the bias is 1, so we start at the D and take three parameters (wrapping around if necessary), which gives us `DPS`.
- Then we append the operations, which is `oonnn`.

Observe that `nn` bitwise-negates the top item on the stack, and then bitwise-negates it again. The two operations cancel out, which means that any `nn` sequence can be optimized out. In practice, these `nn` operations will appear at the end. If you're encoding operations, and you have an even number of leftover operation slots, then you pad out the unused operations with *not*. If you have an odd number of leftover operation slots, then you put *not* in all but the last slot. (Operation 6 is the only one that can be left empty.)

Removing the redundant nn leaves us with `DPSoon`, which matches the value given in the table.

Let's try one of the more complicated operations. Operation index `0xD4` is `0x00D41D78`. The operation code is `0x1D78`:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op5		op4		op3		op2		op1		op6	template			bias	
0	0	0	1	1	1	0	1	0	1	1	1	1	0	0	0
0		1		3		1		1		1	6			0	
n		x		a		x		x		n	SSP*PD			0	

There are four binary operations, so we need five parameters. There is no bias, so we start at the beginning with `SSP`. Next is an asterisk, so we replace it with the first operation, which is `x`. Then we continue with the template, which gives us `PD`. And then we perform the leftover operations, which are `xaxnn`. The resulting RPN is `SSPxPDxaxnn`, which after canceling out the `nn` leaves `SSPxPDxax`. And this matches the value in the table.

If you write a program to apply this algorithm to all of the raster operations, you'll find that decoding the operation code results in the published RPN, with the exception of operation indices 0 and 255.

Let's decode operation index 0, whose operation code is `0x0042`.

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
op5		op4		op3		op2		op1		op6	template			bias	
0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0
0		0		0		0		1		0	0			2	
n		n		n		n		x			SPDD			2	

This decodes as `DDxnnnn`, which simplifies to `DDx`, but the published RPN is just `0`. Aha, but we also know that anything *xor*'d with itself is zero, so `DDx` simplifies further to `0`.

Similarly, operation index 255 has operation code `0x0062` which decodes as follows:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
op5		op4		op3		op2		op1		op6	template			bias	
0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0
0		0		0		0		1		1	0			2	
n		n		n		n		x		n	SPDD			2	

This decodes as `DDxnnnnn`, which simplifies to `DDxn`, and since we learned that `DDx` is `0`, this gives us `0n`, which simplifies further to just `1`, which matches the published RPN.

Back to history: As I noted earlier, the raster opcodes were initially 16-bit values, and the idea was that `BitBlt` implementations would parse and execute the expressions encoded in the operation code. In practice, people just looked up the value in a table of precomputed operation codes and used that to decide how to perform the operation. As a result, the operation index was added to the raster opcode, so that implementations who want to use a table lookup have a single byte to look up instead of having to do a binary search on the 16-bit operation code. Both the operation index and operation code are present so that the values work both with drivers that use lookup tables and drivers which parse the operation code and execute the miniature expression language.

Over time, the drivers that executed the miniature expression language died out. All anybody cares about nowadays is the operation index.

Raymond Chen

Follow