4 Bit Manipulation (bits.hhf)

The HLA BITS module contains several procedures useful for bit manpulation. Currently, this includes routines like counting bits, reversing bits, and merging bit streams.

A Note About Thread Safety: The routines in this module are all thread safe.

Note about stack diagrams: this documentation includes stack diagrams for those functions that pass parameters on the stack. To conserve space, this documentation does not include a stack diagram for any function that does not pass data on the stack (that is, only a return address appears on the stack).

4.1 Bit Module

To call functions in the Bits module, you must include one of the following statements in your HLA application:

```
#include( "bits.hhf" )
or
#include( "stdlib.hhf" )
```

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

4.2 Bit Counting Function

`bits.cnt( b:dword in eax ); @returns( "EAX" );`

This procedure returns the number of one bits present in the "b" parameter (passed in the EAX register). It returns the count in the EAX register. To count the number of zero bits in the parameter value, invert the value of the parameter before passing it to bits.cnt. If you want to count the number of bits in an 8-bit or 16-bit operand, simply zero extend it to 32 bits prior to calling this function. Here are a couple of examples:

If you want to compute the number of bits in an eight-bit operand it's probably faster to write a simple loop that rotates all the bits in the source operand and adds the carry into the accumulating sum. Of course, if performance isn't an issue, you can zero extend the byte to 32 bits and call the bits.cnt procedure.

Note: to count the number of zero bits in an object, first invert than object and then call bits.cnt.

```
HLA high-level calling sequence examples:


bits.cnt( mem32 );
mov( eax, count );

mov( bits.cnt( ebx ), count ); // Note: count is in EAX

// Count the number of bits in 8-bit and 16-bit operands:

bits.cnt( movzx( al, eax ) );
mov( eax, count8 );

mov( bits.cnt( movzx( ax, eax )), count16 ); // Count is left in EAX.



HLA low-level calling sequence examples:

mov( mem32, eax );
call bits.cnt;
mov( eax, count );
```

```
mov( ebx, eax );
call bits.cnt;
mov( eax, count );

movzx( al, eax );
call bits.cnt;
mov( eax, count8 );

movzx( ax, eax );
call bits.cnt;
mov( eax, count16 );
```

4.3 Bit Reversal Functions

The functions in this category swap the bits in their input operand. That is, they exchange the H.O. and L.O. bits, the next-to-high-order bit with bit #1, and so on.

`bits.reverse32( b:dword in eax );  @returns( "eax" );`

This function reverses the bits passed to it in EAX and returns the swapped value in EAX. This function swaps bit 31 and 0, bits 30 and 1, bits 29 and 2, bits 28 and 3, and so on. See the diagram for bits.reverse8 and generalize that diagram to 32 bits for a pictorial example.

```
HLA high-level calling sequence examples:


bits.reverse32( mem32 );
mov( eax, reversed32 );

mov( bits.reverse32( ebx ), ebxReversed ); // Note: result is in EAX



HLA low-level calling sequence examples:

mov( mem32, eax );
call bits.reverse32;
mov( eax, reversed32 );

mov( ebx, eax );
call bits.reverse32;
mov( eax, ebxReversed );
```

`bits.reverse16( b:word in ax );  @returns( "eax" );`

This function reverses the bits passed to it in AX and returns the swapped value in AX. This function swaps bit 15 and 0, bits 14 and 1, bits 13 and 2, bits 12 and 3, and so on. Note that this function does not zero or sign-extend the result into EAX. Although this function currently preserves the H.O. word of EAX, it's safer to assume that the H.O. word of EAX contains an undefined value upon return. See the diagram for bits.reverse8 and generalize that diagram to 16 bits for a pictorial example.

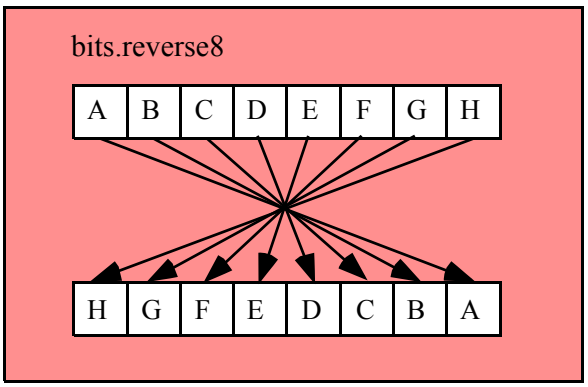
```
HLA high-level calling sequence examples:


bits.reverse16( mem16 );
mov( ax, reversed16 );
```

```
mov( bits.reverse16( bx ), bxReversed ); // Note: result is in AX


HLA low-level calling sequence examples:

mov( mem16, ax );
call bits.reverse16;
mov( ax, reversed16 );

mov( bx, ax );
call bits.reverse16;
mov( ax, bxReversed );
```

bits.reverse8(b:byte in al); @returns("eax");

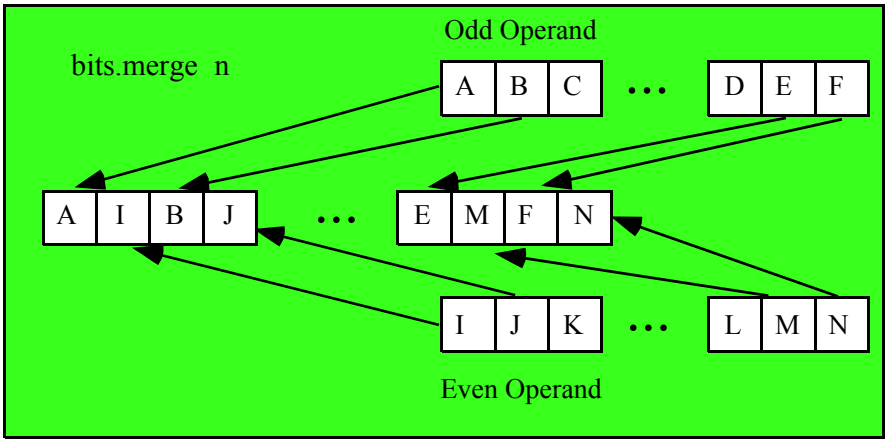
This function reverses the bits passed to it in AL and returns the swapped value in AL. This function swaps bit 7 and 0, bits 6 and 1, bits 5 and 2, and bits 4 and 3 (see the diagram for bits.reverse8). Note that this function does not zero or sign-extend the result into EAX. Although this function currently preserves the H.O. three bytes of EAX, it's safer to assume that the H.O. three bytes of EAX contain an undefined value upon return. See the diagram for bits.reverse8 and generalize that diagram to 16 bits for a pictorial example.

```
HLA high-level calling sequence examples:


bits.reverse8( mem8 );
mov( al, reversed8 );

mov( bits.reverse8( bl ), blReversed ); // Note: result is in AL


HLA low-level calling sequence examples:

mov( mem8, al );
call bits.reverse8;
mov( al, reversed8 );

mov( bl, al );
call bits.reverse8;
mov( al, blReversed );
```



4.4 Bit Merging Functions

The bit merging operands take two small values and produce a single larger value by interleaving the bits from the source operands in the destination operand. One operand's bits are spread out into the destination operand's even bit positions, the other operand's bits are distributed in the odd bit positions (see the following diagram).



`bits.merge32( even:dword in eax; odd:dword in edx );  @returns( "EDX:EAX" );`

This function merges two dword values to produce a single qword value. The bits in the even operand are placed in the even bit positions of the qword result, the bits in the odd operand are merged into the odd bit positions. This function returns the qword result in edx:eax (edx contains the H.O. dword).

Note: because this function passes the parameters in EAX and EDX, you may get an undefined result if you specify EDX as the even parameter or EAX as the odd parameter.

```
HLA high-level calling sequence examples:

bits.merge32( mem32Even, mem32Odd );
mov( edx, (type dword mergedQword[4]) );
mov( eax, (type dword mergedQword[0]) );

bits.merge32( ecx, ebx );
  mov( edx, (type dword mergedQword[4]) );
mov( eax, (type dword mergedQword[0]) );


HLA low-level calling sequence examples:

mov( mem32Even, eax );
mov( mem32Odd, edx );
call bits.merge32;
mov( edx, (type dword mergedQword[4]) );
mov( eax, (type dword mergedQword[0]) );

mov( ecx, eax );
mov( ebx, edx );
call bits.merge32;
mov( edx, (type dword mergedQword[4]) );
mov( eax, (type dword mergedQword[0]) );
```

bits.merge16(even:word; odd:word); @returns("EAX");

This function merges two word values to produce a single dword value. The bits in the even operand are placed in the even bit positions of the dword result, the bits in the odd operand are merged into the odd bit positions. This function returns the dword result in EAX.

```
HLA high-level calling sequence examples:

bits.merge16( mem16Even, mem16Odd );
mov( eax, mergedDword );

bits.merge16( cx, bx );
mov( eax, mergedDword );


HLA low-level calling sequence examples:

  // If mem16Even and mem16Odd are not at the end
  // of some page in memory, you can safely do the following

push( (type dword mem16Even) );
push( (type dword mem16Odd) );
call bits.merge16;
mov( eax, mergedDword );

  // To be absolutely safe, do something like the following
  // (EAX is free to use because the result comes back in EAX):

  movzx( mem16Even, eax );
  push( eax );
  movzx( mem16Odd, eax );
  push( eax );
  call bits.merge16;
  mov( eax, mergedDword );

  push( ecx );
  push( edx );
  call bits.merge16;
  mov( eax, mergedDword );
```

bits.merge8(even:byte in al; odd:byte in ah); @returns("AX");

This function merges two byte values to produce a single word value. The bits in the even operand are placed in the even bit positions of the word result, the bits in the odd operand are merged into the odd bit positions. This function returns the word result in AX.

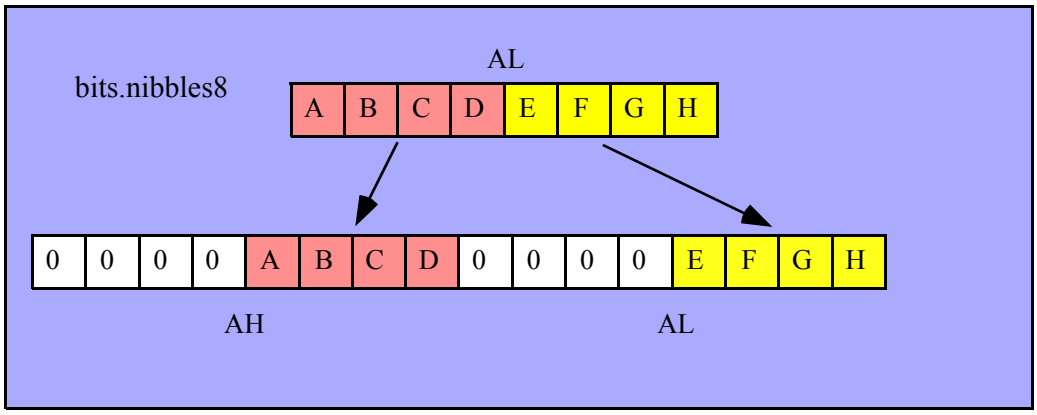
Note: because this function passes the parameters in AL and AH, you may get an undefined result if you specify AH as the even parameter or AL as the odd parameter.

```
HLA high-level calling sequence examples:

bits.merge8( mem8Even, mem8Odd );
mov( ax, mergedWord );

bits.merge8( cl, bh );
mov( eax, mergedWord );


HLA low-level calling sequence examples:
```

```
mov( mem8Even, al );
mov( mem8Odd, ah );
call bits.merge8;
mov( ax, mergedWord );

mov( cl, al );
mov( bh, ah );
call bits.merge8;
mov( ax, mergedWord );
```

4.5 Bit Extraction Functions

These functions extract nibbles from their source operands, zero extend those nibbles to bytes, and return the values in an operand twice the size of the original operand (e.g., AL->AX, AX->EAX, and EAX->EDX:EAX). For example, the bits.nibbles8 function does the following:



`bits.nibbles32( d:dword in eax );  @returns( "EDX:EAX" );`

This function extracts the 8 nibbles from EAX, zero extends each of them to 8 bits, and then places them in the following locations

```
EAX[0:3]   -> EAX[0:7]
EAX[4:7]   -> EAX[8:15]
EAX[8:11]  -> EAX[16:23]
EAX[12:15] -> EAX[24:31]
EAX[16:19] -> EDX[0:7]
EAX[20:23] -> EDX[8:15]
EAX[24:27] -> EDX[16:23]
EAX[28:31] -> EDX[24:31]
```

```
HLA high-level calling sequence examples:

bits.nibbles32( mem32 );
mov( eax, LONibbles );
mov( edx, HONibbles );

bits.nibbles32( ecx );
mov( eax, LONibbles );
mov( edx, HONibbles );
```

```
HLA low-level calling sequence examples:

mov( mem32, eax );
call bits.nibbles32;
mov( ax, mergedWord );

mov( ecx, eax );
call bits.nibbles32;
mov( eax, LONibbles );
mov( edx, HONibbles );
```

bits.nibbles16(w:word in ax); @returns("EAX");

This function extracts the 4 nibbles from AX, zero extends each of them to 8 bits, and then places them in the following locations

```
AX[0:3]   -> EAX[0:7]
AX[4:7]   -> EAX[8:15]
AX[8:11]  -> EAX[16:23]
AX[12:15] -> EAX[24:31]

HLA high-level calling sequence examples:

bits.nibbles16( mem16 );
mov( eax, Nibbles );

bits.nibbles16( cx );
mov( eax, Nibbles );


HLA low-level calling sequence examples:

mov( mem16, ax );
call bits.nibbles16;
mov( ax, mergedWord );

mov( cx, ax );
call bits.nibbles16;
mov( eax, LONibbles );
```

bits.nibbles8(b:byte in al); @returns("AX");

This function extracts the 2 nibbles from AL, zero extends each of them to 8 bits, and then places them in the following locations

```
AL[0:3] -> AL[0:7]
AL[4:7] -> AH[0:7]


HLA high-level calling sequence examples:

bits.nibbles8( mem8 );
mov( ax, TwoNibbles );

bits.nibbles8( ch );
mov( ax, TwoNibbles );
```

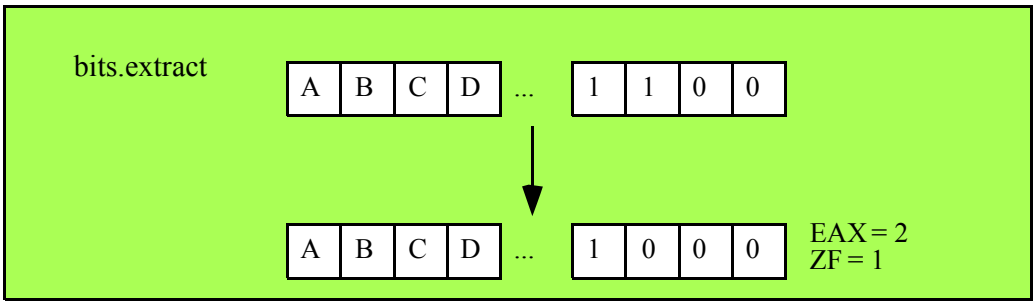
```
HLA low-level calling sequence examples:

mov( mem8, al );
call bits.nibbles8;
mov( ax, TwoNibbles );

mov( ch, al );
call bits.nibbles8;
mov( ax, TwoNibbles );
```

procedure bits.extract(var d:dword);

@returns("EAX"); // Really a macro.

This function extracts the first set bit in d searching from bit #0 and returns the index of this bit in the EAX register; the function will also return the zero flag clear in this case. This function also clears that bit in the operand. If d contains zero, then this function returns the zero flag set and EAX will contain -1.

Note that HLA actually implements this function as a macro, not a procedure. This means that you can pass any double word operand as a parameter (i.e., a memory or a register operand). However, the results are undefined if you pass EAX as the parameter (since this function returns the bit number in EAX). Note that the macro will report an error message if you try to pass EAX as the parameter.



4.6 Bit Distribution Functions

bits.distribute(source:dword; mask:dword; dest:dword);

@returns("EAX");

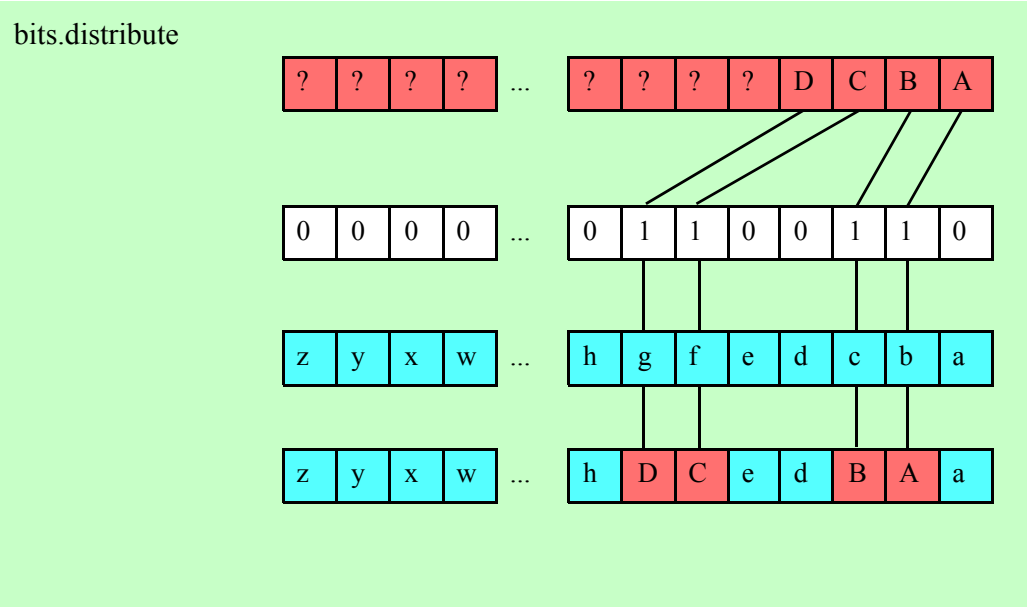
This function takes the L.O. *n* bits of *source*, where *n* is the number of "1" bits in mask, and inserts these bits into dest at the bit positions specified by the "1" bits in mask. This function does not change the bits in dest that correspond to the zeros in the mask value. This function does not affect the value of the actual dest parameter, instead, it returns the new value in the EAX register.

Example:

```
source = $FF00_AA55
mask = $F0FF_000F
dest = $1234_5678
```

The bits.distribute function grabs the L.O. 16 bits of source and inserts these bits at positions 0, 1, 2, 3, 16, 17, 18, 19, 20, 21, 22, 23, and 28, 29, 30, and 31 into the value $1234_5678 and returns the result in EAX. For this example, bits.distribute begins by grabbing the L.O. four bits of source and inserts them at bit positions 0..3 of $1234_5678 (since mask contains four set bits at positions 0..3). This yields the temporary result

$1234_5675. Next, bits.distribute grabs bits 4..11 from source ($A5) and inserts these bits into bit positions 16..23 since mask contains eight consecutive bits between positions 16 and 23. The produces the temporary result $12A5_5675. Finally, bits.distribute grabs bits 12..15 from the source operand ($A) and inserts these into the result at bit positions 28..31 (since the mask value contains set bits at these positions). The final result this function returns in EAX is $A2A5_5675.




```
bits.coalese( source:dword; mask:dword );
```

```
    @returns( "EAX" );
```

This function is the converse of bits.distribute. It extracts all the bits in source whose corresponding positions in mask contain a one. This function coalesces (right justifies) these bits in the L.O. bit positions of the result and returns the result in EAX.

Example:

```
source = $afff_ffce
mask = $aaaa_5555
```

bits.coalesce grabs bits 0, 2, 4, 6, 8, 10, 12, 14, 17, 19, 21, 23, 25, 27, 29, and 31 from source and packs these into the L.O. 16 bits of EAX (it also sets the H.O. bits of EAX to zero). The final result in EAX is $FFFA.