31 The Standard Output Module (stdout.hhf)

This unit contains routines that write data to the standard output device. This is usually the console device, although the user may redirect the standard output to a file from the command line.

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

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).

A Note About Thread Safety: Because the standard output device is a single resource, you will get inconsistent results if multiple threads attempt to write to the standard output device simultaneously. The HLA standard library stdout module does not attempt to synchronize thread access to the standard output device. If you are going to be writing to the standard output from multiple threads, it is your responsibility to ensure that the threads use properly synchronized access to this resource.

A Note About the FPU: The Standard Library code makes occasional use of the FPU, particularly when converting between real and string formats and when computung certain mathematical functions. You should exercise caution when using MMX instructions in a program that makes use of the Standard Library. In particular, you should ensure that you are always in FPU mode (by executing an EMMS instruction) after you are finished using MMX instructions. Better yet, you should avoid the MMX instruction set altogether and use the improved SSE instruction set that accomplishes the same tasks (and doesn't disturb the FPU).

31.1 Conversion Format Control

The standard output functions that convert numeric values to hexadecimal, unsigned decimal, and signed decimal output provide the ability to inject underscores between groups of three (decimal) or four (hexadecimal) digits to make it easier to read large numbers. You enable and disable underscore output using the conv.setUnderscores and conv.getUnderscores functions. Please refer to their documentation in the conv.rtf file for more details.

When converting numeric values to string form for output, the standard output routines call the conversion functions found in the conv (conversions) module. For detailed information on the actual conversions, please consult the conv.rtf document.

31.2 File I/O Routines and the Standard Output Handle

The standard output routines are basically a thin layer over the top of the fileio routines (see the fileio documention for a complete description of those routines). Indeed, if you obtain the standard output handle, you can write data to the standard output device by passing this handle to a fileio function. Because the fileio module provides a slightly richer set of routines, there are a few instances where you might want to do this. You might also want to write a generic output function that expects a file handle and then pass it the standard output device file handle so that the function writes its output to the console (or other standard output device) rather than to some file. In any case, just be aware that it is perfectly reasonable to call fileio functions to write data to the standard output device.

```
stdout.handle; @returns( "eax" );
```

This routine returns the Linux/Windows handle for the Standard Output Device in the EAX register. You may use this handle with the file I/O routines to write data to the standard output device.

31.3 Standard Output Routines

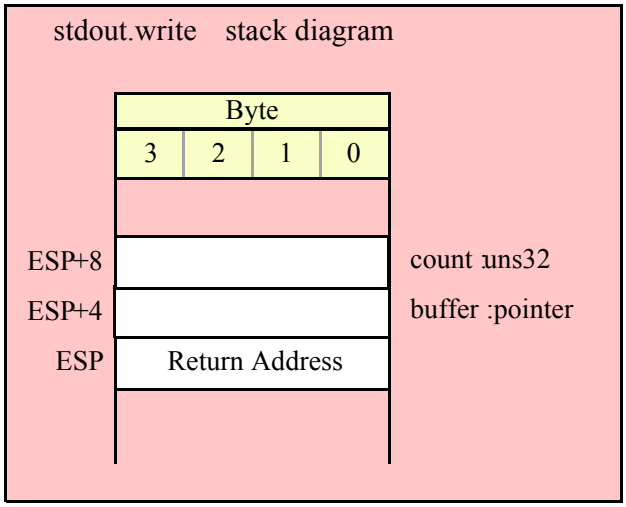
The output routines in the stdout module are very similar to the file output routines in the stdout module. In general, these routines require (at least) one parameter: the value to write to the standard output. Some functions contain additional parameters that provide formatting information.

31.4 Miscellaneous Output Routines

stdout.write(var buffer:var; count:uns32);

This procedure writes the number of bytes specified by the count variable to the standard output device. The bytes starting at the address of the buffer variable are written to the standard out. No range checking is done on the buffer, it is your responsibility to ensure that the buffer contains at least count valid data bytes. Because the buffer parameter is passed by untyped reference, a high-level style call to this function will take the address of whatever object you supply as the buffer parameter. *This includes pointer variables* (which is probably not what you want to do). Use the VAL keyword in a high-level style call if you want to use the value of a pointer variable rather than the address of that pointer variable (see the examples that follow).

```
HLA high-level calling sequence examples:

stdout.write( buffer, count );

// If "bufPtr" is dword containing the address of the buffer, then
// use the following code:

stdout.write( val bufPtr, bufferSize );

// If you actually want to write out the four bytes held by
// bufPtr (an unusual thing to do), you would use the
// following code:

stdout.write( bufPtr, 4 );



HLA low-level calling sequence examples:

// Assumes buffer is a static object at a fixed
// address in memory:

pushd( &buffer );
push( count );
call stdout.write;

  // If a 32-bit register is available and buffer
  // isn't at a fixed, static, address:

lea( eax, buffer );
push( eax );
push( count );
call stdout.write;

  // If a 32-bit register is not available and buffer
  // isn't at a fixed, static, address:

sub( 4, esp );
push( eax );
lea( eax, buffer );
mov( eax, [esp+4] );
pop( eax );
push( count );
call stdout.write;
```



`stdout.newln()`

This function writes a newline sequence (e.g., carriage return/line feed under Windows or line feed under Linux) to the output.

```
HLA high-level calling sequence examples:

stdout.newln();


HLA low-level calling sequence examples:


call stdout.newln;
```

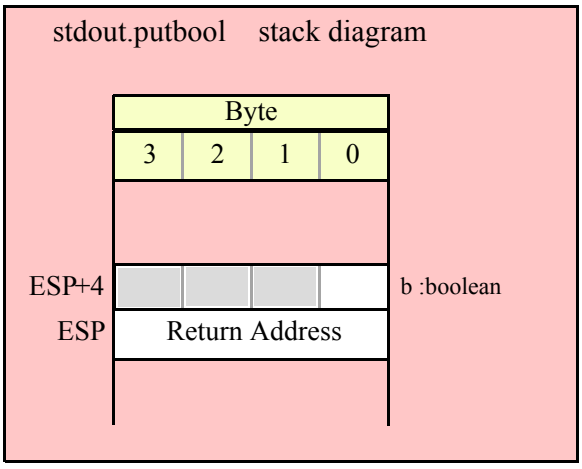
31.5 Boolean Output

`stdout.putbool( b:boolean );`

This procedure writes the string "true" or "false" to the standard output depending on the value of the b parameter.

```
HLA high-level calling sequence examples:

stdout.putbool( boolVar );

// If the boolean is in a register (AL):

stdout.putbool( al );


HLA low-level calling sequence examples:


  // If "boolVar" is not one of the last three
  // bytes on a page of memory, you can do this:
```

```
push( (type dword boolVar ) );
call stdout.putbool;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( boolVar , eax ); // Assume EAX is available
push( eax );
call stdout.putbool;

// If no register is available, do something
// like the following code:

sub( 4, esp );
push( eax );
movzx( boolVar , eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.putbool;

// If the boolean value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax );  // Assume boolVar is in AL
call stdout.putbool;

// If the Boolean value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume boolVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stdout.putbool;
```

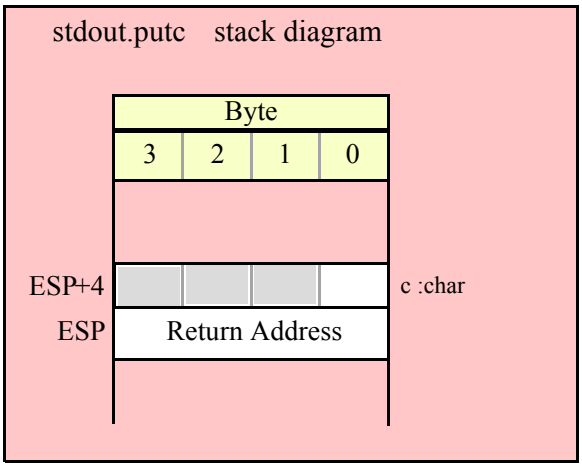


31.6 Character, String, and Character Set Output Routines

`stdout.putc( c:char );`

Writes the character specified by the c parameter to the standard output device.

```
HLA high-level calling sequence examples:

stdout.putc( charVar );

// If the character is in a register (AL):

stdout.putc( al );


HLA low-level calling sequence examples:


  // If "charVar" is not one of the last three
  // bytes on a page of memory, you can do this:

push( (type dword charVar) );
call stdout.putc;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( charVar, eax ); // Assume EAX is available
push( eax );
call stdout.putc;

// If no register is available, do something
// like the following code:

sub( 4, esp );
push( eax );
movzx( charVar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.putc;

// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax );  // Assume charVar is in AL
call stdout.putc;

// If the character value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume charVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stdout.putc;
```

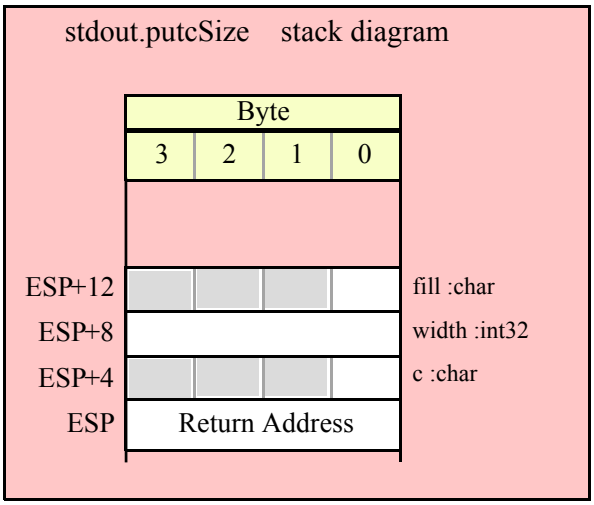


stdout.putcSize(c:char; width:int32; fill:char)

Outputs the character c to the standard output using at least width output positions. If the absolute value of width is greater than one, then this function writes fill characters as padding characters during the output. If width is a positive value greater than one, then stdout.putcSize writes c left justfied in a field of width characters; if width is a negative value less than one, then stdout.putcSize writes c right justified in a field of width characters.

```
HLA high-level calling sequence examples:

stdout.putcSize( charVar, width, padChar );


HLA low-level calling sequence examples:


  // If "charVar" and "padChar" are not one of the last three
  // bytes on a page of memory, you can do this:

push( (type dword charVar) );
push( width );
push( (type dword padChar) );
call stdout.putcSize;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( charVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putcSize;

// If no registers are available, do something
// like the following code:

push( eax );
movzx( charVar, eax );
push( eax );
push( width );
```

```
movzx( padChar, eax );
push( eax );
call stdout.putcSize;
pop( eax );

// If "charVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax );   // Assume charVar is in AL
push( width );
push( ebx );   // Assume padChar is in BL
call stdout.putcSize;

// Do the following if the characters are
// in AH, BH, CH, or DH:

xchg( al, ah );    // Assume charVar is in AH
xchg( bl, bh );    // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call stdout.putcSize;
```

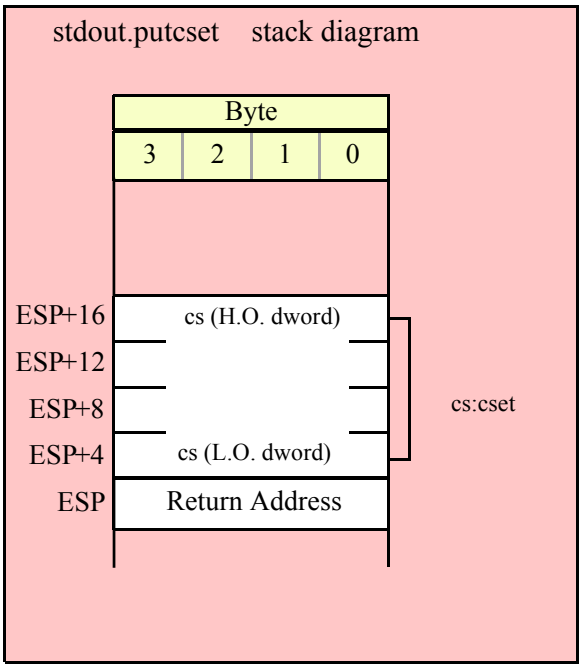


stdout.putcset(cst:cset);

This function writes all the members of the cst character set parameter to the standard output device.

```
HLA high-level calling sequence examples:

stdout.putcset( csVar );
stdout.putcset( [ebx] ); // EBX points at the cset.


HLA low-level calling sequence examples:

push( (type dword csVar[12]) );  // Push H.O. dword first
push( (type dword csVar[8]) );
push( (type dword csVar[4]) );
```

```
push( (type dword csVar) );      // Push L.O. dword last
call stdout.putcset;

push( (type dword [ebx+12]) );   // Push H.O. dword first
push( (type dword [ebx+8]) );
push( (type dword [ebx+4]) );
push( (type dword [ebx]) );      // Push L.O. dword last
call stdout.putcset;
```

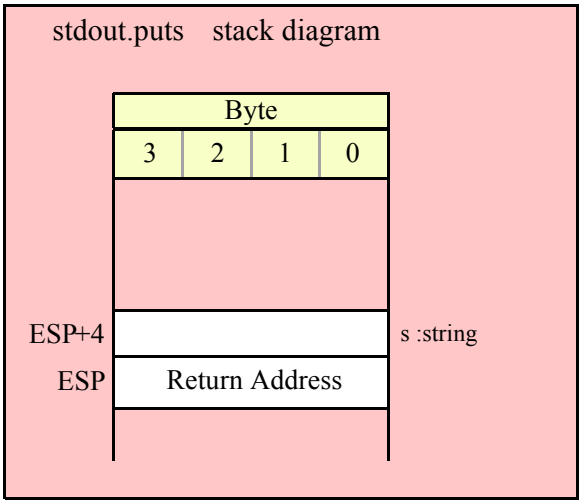


`stdout.puts( s:string );`

This procedure writes the value of the string parameter to the standard output. Remember, string values are actually 4-byte pointers to the string's character data.

```
HLA high-level calling sequence examples:

stdout.puts( strVar );
stdout.puts( ebx ); // EBX holds a string value.
stdout.puts( "Hello World" );


HLA low-level calling sequence examples:

// For string variables:

push( strVar );
call stdout.puts;

// For string values held in registers:

push( ebx );  // Assume EBX holds the string value
call stdout.puts;

// For string literals, assuming a 32-bit register
// is available:
```

```
lea( eax, "Hello World" ); // Assume EAX is available.
push( eax );
call stdout.puts;

// If a 32-bit register is not available:

readonly
  literalString :string := "Hello World";
      .
      .
      .
push( literalString );
call stdout.puts;
```

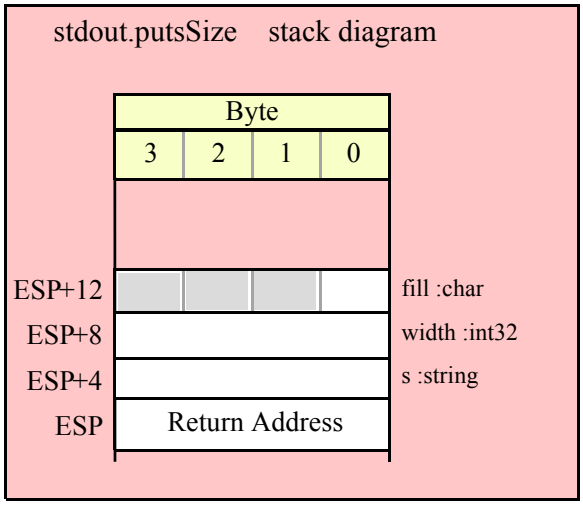


`stdout.putsSize( s:string; width:int32; fill:char );`

This function writes the s string to the standard output using at least width character positions. If the absolute value of width is less than or equal to the length of s, then this function behaves exactly like stdout.puts. On the other hand, if the absolute value of width is greater than the length of s, then stdout.putsSize writes width characters to the standard output. This procedure emits the fill character in the extra print positions. If width is positive, then stdout.putsSize right justifies the string in the print field. If width is negative, then stdout.putsSize left justifies the string in the print field. Generally, people expect the string to be left justified, so you should ensure that this value is negative to achieve this.

```
HLA high-level calling sequence examples:

stdout.putsSize( strVar, width, ' ' );

// For the following, EBX holds the string value,
// ECX contains the width, and AL holds the pad
// character:

stdout.putsSize( ebx, ecx, al );

stdout.putsSize( "Hello World", 25, padChar );


HLA low-level calling sequence examples:

// For string variables:
```

```
push( strVar );
push( width );
pushd( ' ' );
call stdout.putsSize;

// For string values held in registers:

push( ebx );  // Assume EBX holds the string value
push( ecx );  // Assume ECX holds the width
push( eax );  // Assume AL holds the fill character
call stdout.putsSize;

// For string literals, assuming a 32-bit register
// is available:

lea( eax, "Hello World" ); // Assume EAX is available.
push( eax );
pushd( 25 );
movzx( padChar, eax );
push( eax );
call stdout.putsSize;

// If a 32-bit register is not available:

readonly
  literalString :string := "Hello World";

  // Note: element zero is the actual pad character.
   // The other elements are just padding.
   padChar :char[4] := [ '.', #0, #0, #0 ];
      .
      .
      .
push( literalString );
pushd( 25 );
push( (type dword padChar) );
call stdout.putsSize;
```



31.7 Hexadecimal Output Routines

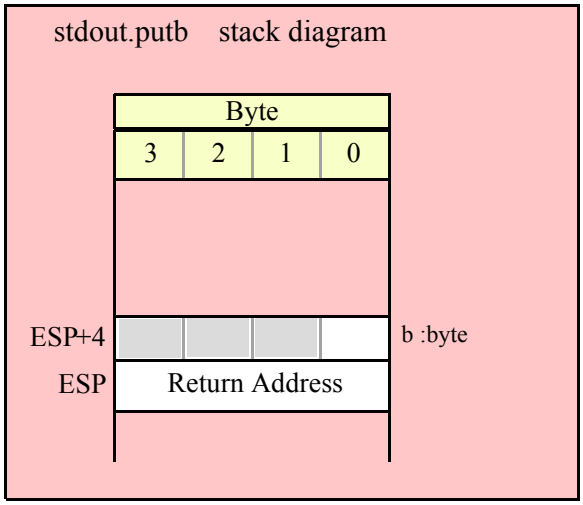
These routines convert numeric data to hexadecimal string form (using the hexadecimal conversion routines found in the conv module) and write the resulting string to the standard output device.

stdout.putb(b:byte)

This procedure writes the value of b to the standard output using exactly two hexadecimal digits (including a leading zero if necessary).

```
HLA high-level calling sequence examples:


stdout.putb( byteVar );

// If the character is in a register (AL):

stdout.putb( al );



HLA low-level calling sequence examples:


  // If "byteVar " is not one of the last three
  // bytes on a page of memory, you can do this:

push( (type dword byteVar ) );
call stdout.putb;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call stdout.putb;

// If no register is available, do something
// like the following code:

sub( 4, esp );
push( eax );
movzx( byteVar , eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.putb;

// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax );  // Assume byteVar is in AL
call stdout.putb;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
```

```
call stdout.putb;
```

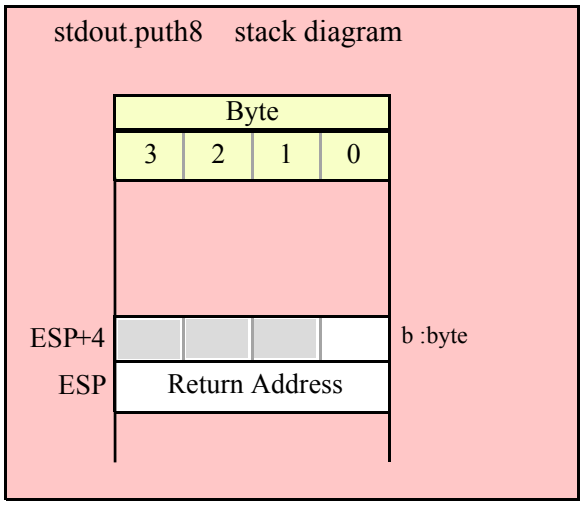


`stdout.puth8( b:byte );`

This procedure writes the value of b to the standard output using the minimum necessary number of hexadecimal digits.

```
HLA high-level calling sequence examples:


stdout.puth8( byteVar );

// If the character is in a register (AL):

stdout.puth8( al );



HLA low-level calling sequence examples:


  // If "byteVar " is not one of the last three
  // bytes on a page of memory, you can do this:

push( (type dword byteVar ) );
call stdout.puth8;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call stdout.puth8;

// If no register is available, do something
// like the following code:

sub( 4, esp );
push( eax );
```

```
movzx( byteVar , eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.puth8;

// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax );  // Assume byteVar is in AL
call stdout.puth8;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stdout.puth8;
```

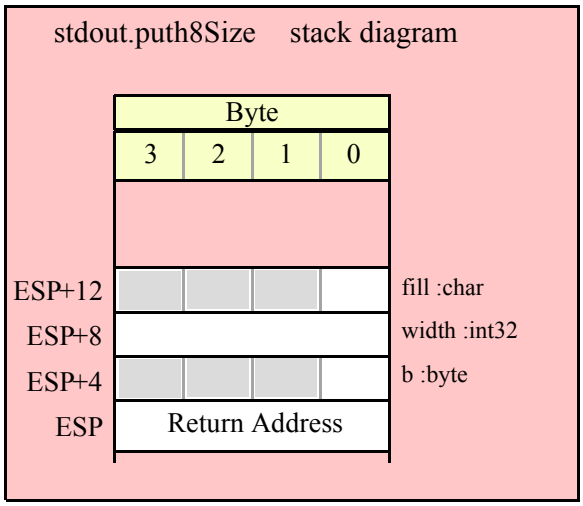


stdout.puth8Size(b:byte; size:dword; fill:char)

The stdout.puth8Size function writes an 8-bit hexadecimal value to the standard output allowing you specify a minimum field width and a fill character.

```
HLA high-level calling sequence examples:

stdout.puth8Size( byteVar, width, padChar );


HLA low-level calling sequence examples:


  // If "byteVar" and "padChar" are not one of the last three
  // bytes on a page of memory, you can do this:

push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call stdout.puth8Size;

// If you can't guarantee that the previous code
```

```
    // won't generate an illegal memory access, and a
    // 32-bit register is available, use code like
    // the following:

    movzx( byteVar, eax ); // Assume EAX is available
    push( eax );
    push( width );
    movzx( padChar, ebx ); // Assume EBX is available
    push( ebx );
    call stdout.puth8Size;

    // If no registers are available, do something
    // like the following code:

    push( eax );
    movzx( byteVar, eax );
    push( eax );
    push( width );
    movzx( padChar, eax );
    push( eax );
    call stdout.puth8Size;
    pop( eax );

    // If "byteVar" or "padChar" are in an
    // 8-bit register, then you can push
    // the corresponding 32-bit register if
    // the register is AL, BL, CL, or DL:

    push( eax );   // Assume byteVar is in AL
    push( width );
    push( ebx );   // Assume padChar is in BL
    call stdout.puth8Size;

    // Do the following if the characters are
    // in AH, BH, CH, or DH:

    xchg( al, ah );    // Assume byteVar is in AH
    xchg( bl, bh );    // Assume padChar is in BH
    push( eax );
    push( width );
    push( ebx );
    xchg( al, ah );
    xchg( bl, bh );
    call stdout.puth8Size;
```

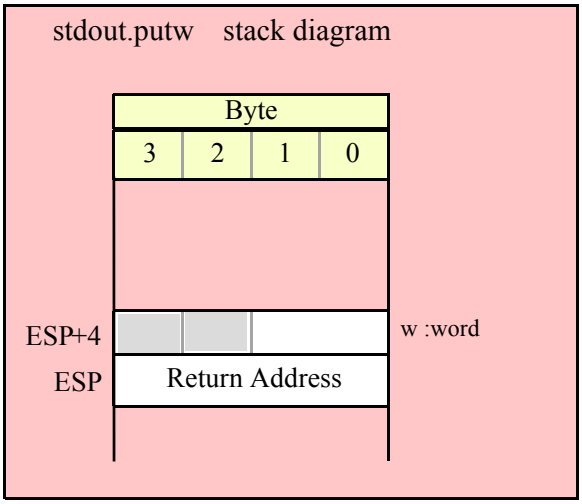


stdout.putw(w:word)

This procedure writes the value of w to the standard output device using exactly four hexadecimal digits (including leading zeros if necessary).

```
HLA high-level calling sequence examples:


stdout.putw( wordVar );

// If the word is in a register (AX):

stdout.putw( ax );



HLA low-level calling sequence examples:


  // If "wordVar " is not one of the last three
  // bytes on a page of memory, you can do this:

push( (type dword wordVar) );
call stdout.putw;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call stdout.putw;

// If no register is available, do something
// like the following code:

push( eax ):
movzx( wordVar, eax );
push( eax );
call stdout.putw;
pop( eax );
```

```
// If the word value is in a 16-bit register
// then you can use code like the following:

push( eax );  // Assume wordVar is in AX
call stdout.putw;
```

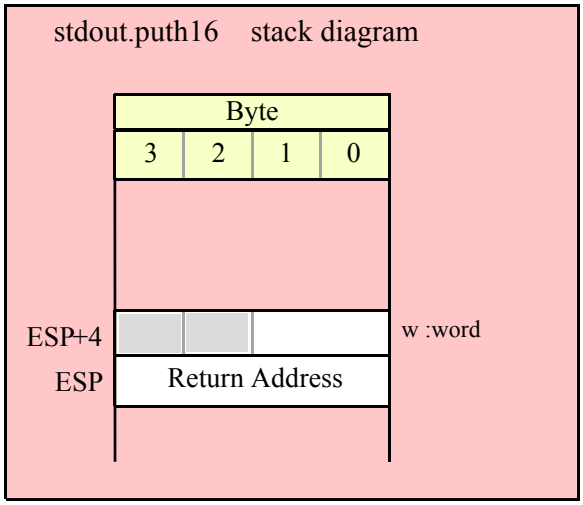


stdout.puth16(w:word)

This procedure writes the value of w to the standard out using the minimum necessary number of hexadecimal digits.

```
HLA high-level calling sequence examples:


stdout.puth16( wordVar );

// If the word is in a register (AX):

stdout.puth16( ax );



HLA low-level calling sequence examples:


  // If "wordVar " is not one of the last three
  // bytes on a page of memory, you can do this:

push( (type dword wordVar) );
call stdout.puth16;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call stdout.puth16;

// If no register is available, do something
```

```
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
call stdout.puth16;
pop( eax );

// If the word value is in a 16-bit register
// then you can use code like the following:

push( eax );  // Assume wordVar is in AX
call stdout.puth16;
```

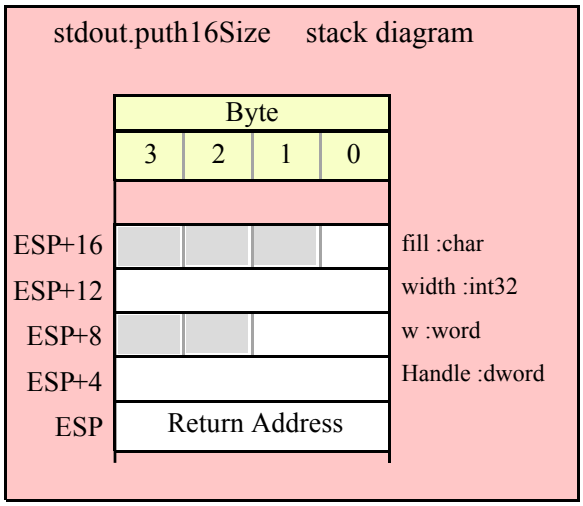


stdout.puth16Size(w:word; size:dword; fill:char)

The stdout.puth16Size function writes a 16-bit hexadecimal value to the standard output allowing you specify a minimum field width and a fill character.

```
HLA high-level calling sequence examples:

stdout.puth16Size( wordVar, width, padChar );


HLA low-level calling sequence examples:


  // If "wordVar" and "padChar" are not one of the last three
  // bytes on a page of memory, you can do this:

push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call stdout.puth16Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( wordVar, eax ); // Assume EAX is available
push( eax );
```

```
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puth16Size;

// If no registers are available, do something
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call stdout.puth16Size;
pop( eax );

// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax );   // Assume wordVar is in AX
push( width );
push( ebx );   // Assume padChar is in BL
call stdout.puth16Size;
```

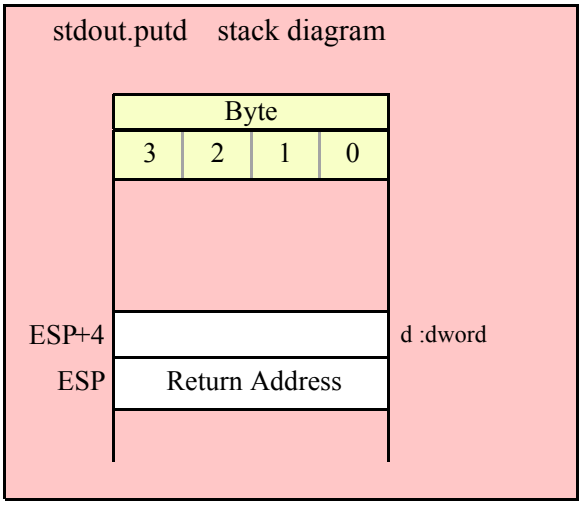


stdout.putd(d:dword)

This procedure writes the value of d to the standard out using exactly eight hexadecimal digits (including leading zeros if necessary), if underscore output is not enabled. This routine will emit nine characters (eight digits plus an underscore) if underscore output is enabled.

```
HLA high-level calling sequence examples:


stdout.putd( dwordVar );

// If the dword value is in a register (EAX):
```

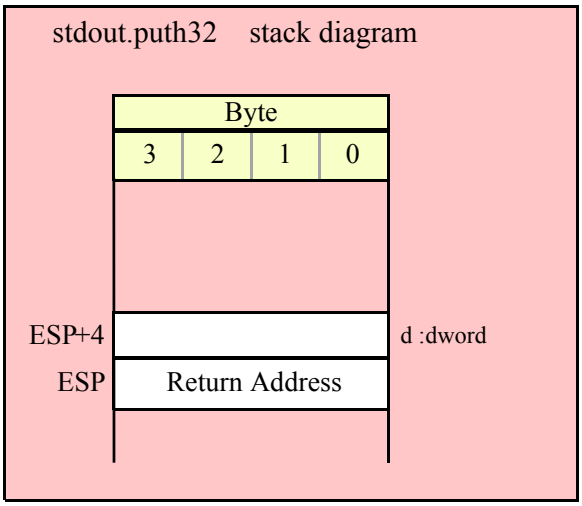
```
stdout.putd( eax );


HLA low-level calling sequence examples:

push( dwordVar );
call stdout.putd;

push( eax );
call stdout.putd;
```



stdout.puth32(d:dword);

This procedure writes the value of d to the standard output using the minimum number of hexadecimal digits necessary. If underscore output is enabled (see conv.setUnderscores and conv.getUnderscores) then this function will emit an underscore between groups of four hexadecimal digits, starting from the least signficant digit.

```
HLA high-level calling sequence examples:


stdout.puth32( dwordVar );

// If the dword is in a register (EAX):

stdout.puth32( eax );


HLA low-level calling sequence examples:

push( dwordVar );
call stdout.puth32;

push( eax );
call stdout.puth32;
```

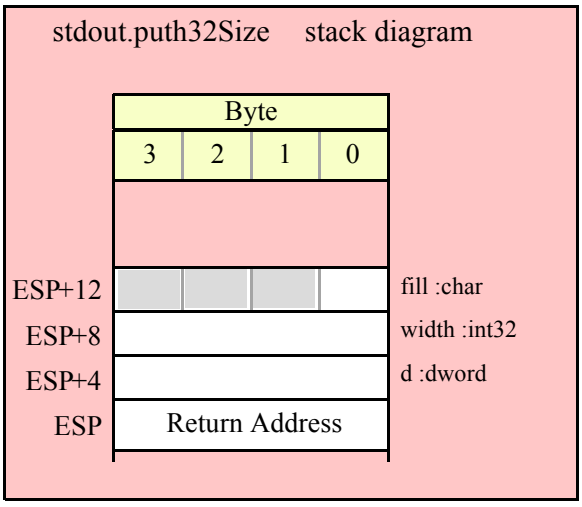


stdout.puth32Size(d:dword; size:dword; fill:char)

The stdout.puth32Size function outputs d as a hexadecimal string (including underscores, if enabled) and it allows you specify a minimum field width and a fill character.

```
HLA high-level calling sequence examples:


stdout.puth32Size( dwordVar, width, ' ' );

// If the dword is in a register (EAX):

stdout.puth32Size( eax, width, cl );



HLA low-level calling sequence examples:

push( dwordVar );
push( width );
pushd( ' ' );
call stdout.puth32Size;

push( eax );
push( width );
push( ecx ); // fill char is in CL
call stdout.puth32Size;

// Assume fill char is in CH

push( eax );
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.puth32Size;

// Alternate method of the above

push( eax );
push( width );
```

```
    sub( 4, esp );
    mov( ch, [esp] );
    call stdout.puth32Size;

    // If the fill char is a variable and
    // a register is available, try this code:

    push( eax );
    push( width );
    movzx( fillChar, ebx ); // Assume EBX is available
    push( ebx );
    call stdout.puth32Size;


    // If the fill char is a variable and
    // no register is available, here's one
    // possibility:

    push( eax );
    push( width );
    push( (type dword fillChar) ); // Chance of page crossing!
    call stdout.puth32Size;

    // In the very rare case that the above would
    // cause an illegal memory access, use this:

    push( eax );
    push( width );
    sub( 4, esp );
    push( eax );
    movzx( fillChar, eax );
    mov( eax, [esp+4] );
    pop( eax );
    call stdout.puth32Size;
```

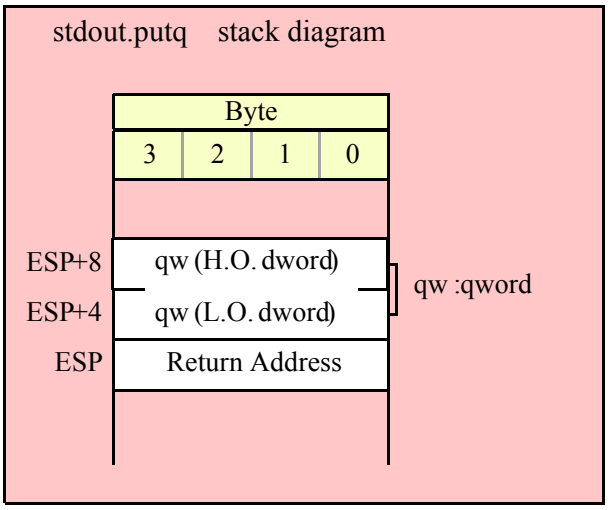


`stdout.putq( q:qword );`

This procedure writes the value of q to the standard output device using exactly sixteen hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

```
    HLA high-level calling sequence examples:
```

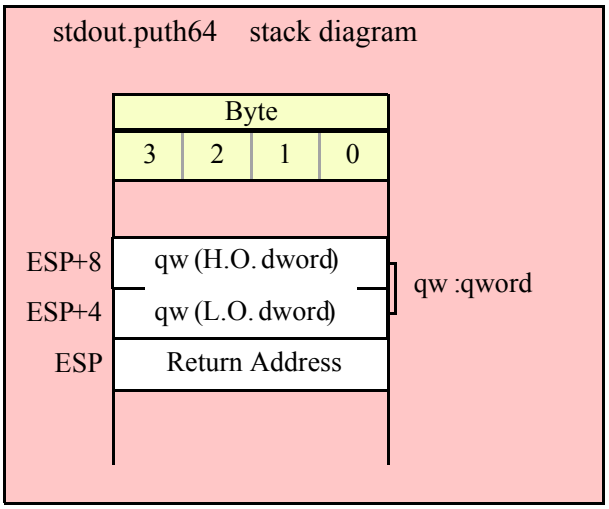
```
stdout.putq( qwordVar );


HLA low-level calling sequence examples:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
call stdout.putq;
```



`stdout.puth64( q:qword );`

This procedure writes the value of q to the standard output using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

```
HLA high-level calling sequence examples:


stdout.puth64( qwordVar );


HLA low-level calling sequence examples:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
call stdout.puth64;
```

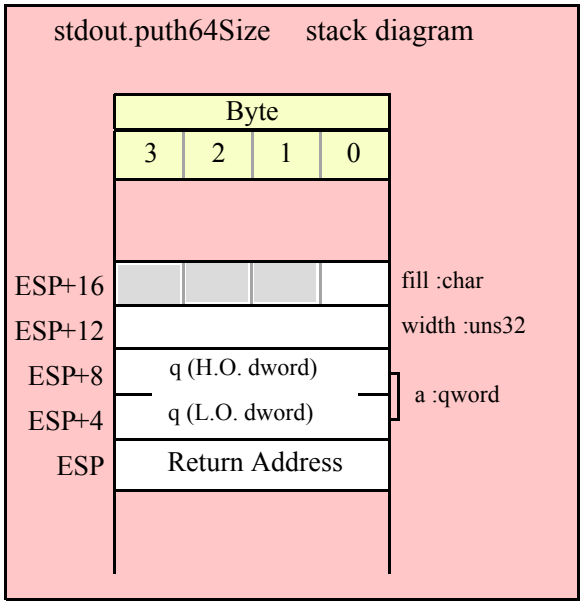


stdout.puth64Size(q:qword; size:dword; fill:char);

The stdout.putqSize function lets you specify a minimum field width and a fill character. The stdout.putq routine uses a minimum size of two and a fill character of '0'. Note that if underscore output is enabled, this routine will emit 19 characters (16 digits plus three underscores).

```
HLA high-level calling sequence examples:


stdout.puth64Size( qwordVar, width, ' ' );



HLA low-level calling sequence examples:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
pushd( ' ' );
call stdout.puth64Size;



push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call stdout.puth64Size;

// Assume fill char is in CH

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.puth64Size;

// Alternate method of the above
```

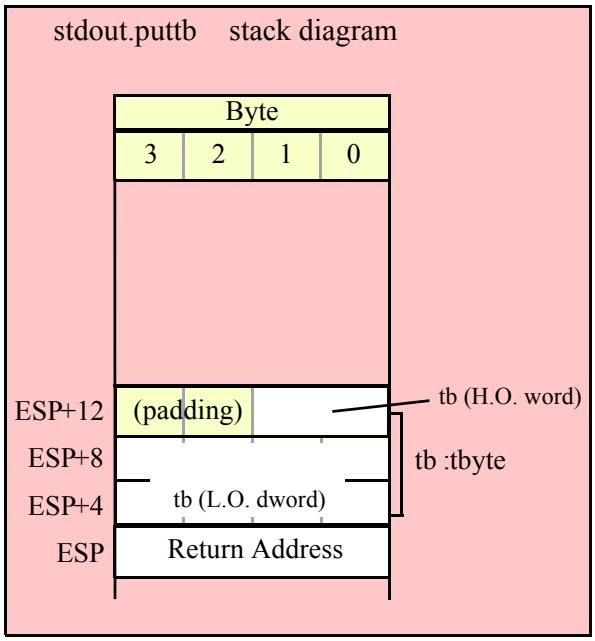
```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.puth64Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puth64Size;


// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stdout.puth64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.puth64Size;
```



`stdout.puttb( tb:tbyte );`

This procedure writes the value of tb to the standard out using exactly 20 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

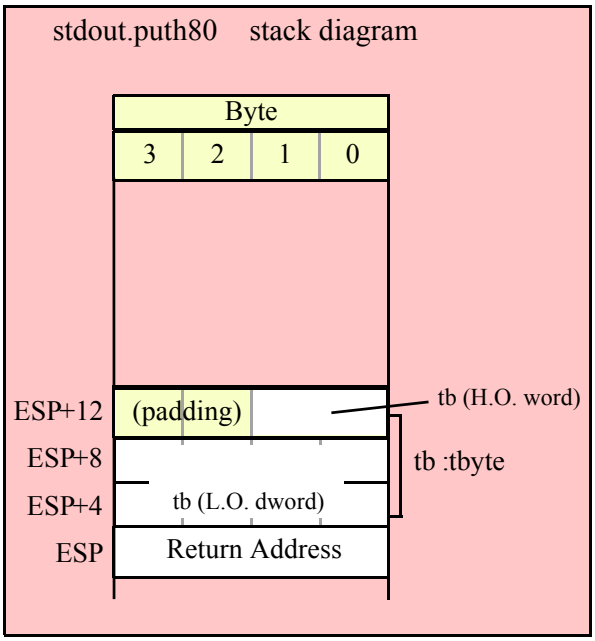
```
HLA high-level calling sequence examples:


stdout.puttb( tbyteVar );



HLA low-level calling sequence examples:

pushw( 0 );                  // Push push a 0 pad word
push( (type word tbyteVar[8]));   // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));     // L.O. dword last
call stdout.puttb;
```



`stdout.puth80( tb:tbyte );`

This procedure writes the value of tb to the standard output using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

```
HLA high-level calling sequence examples:


stdout.puth80( tbyteVar );



HLA low-level calling sequence examples:

pushw( 0 );                  // Push push a 0 pad word
push( (type word tbyteVar[8]));   // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));     // L.O. dword last
call stdout.puth80;
```

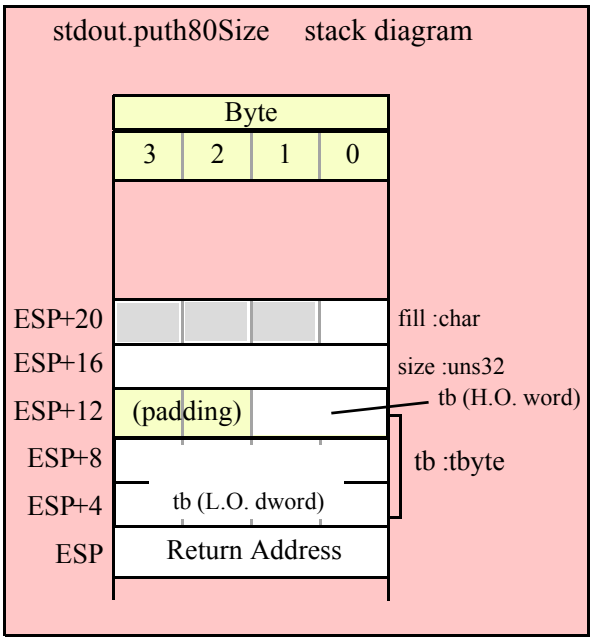


`stdout.puth80Size( tb:tbyte; size:dword; fill:char );`

The stdout.puth80Size function lets you specify a minimum field width and a fill character. It writes the tbyte value tb as a hexadecimal string to the standard output device using the provided minimum size and fill character.

```
HLA high-level calling sequence examples:


stdout.puth80Size( tbyteVar, width, ' ' );



HLA low-level calling sequence examples:

pushw( 0 );                  // Push push a 0 pad word
push( (type word tbyteVar[8]));   // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));     // L.O. dword last
push( width );
pushd( ' ' );
call stdout.puth80Size;



// Assume fill char is in CH

pushw( 0 );                  // Push push a 0 pad word
push( (type word tbyteVar[8]));   // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));     // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.puth80Size;
```

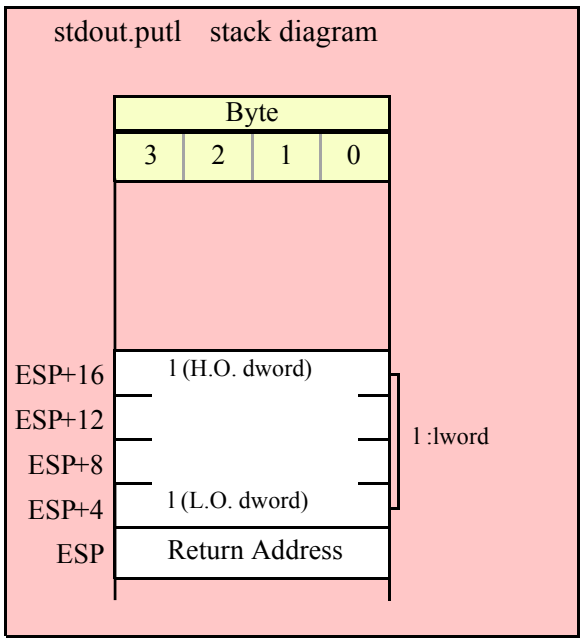
```
// Alternate method of the above

pushw( 0 );                  // Push push a 0 pad word
push( (type word tbyteVar[8]));   // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));     // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.puth80Size;

// If the fill char is a variable and
// a register is available, try this code:

pushw( 0 );                  // Push push a 0 pad word
push( (type word tbyteVar[8]));   // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));     // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puth80Size;


// If the fill char is a variable and
// no register is available, here's one
// possibility:

pushw( 0 );                  // Push push a 0 pad word
push( (type word tbyteVar[8]));   // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));     // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stdout.puth80Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

pushw( 0 );                  // Push push a 0 pad word
push( (type word tbyteVar[8]));   // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));     // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.puth80Size;
```



`stdout.putl( l:lword );`

This procedure writes the value of l to the standard output using exactly 32 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

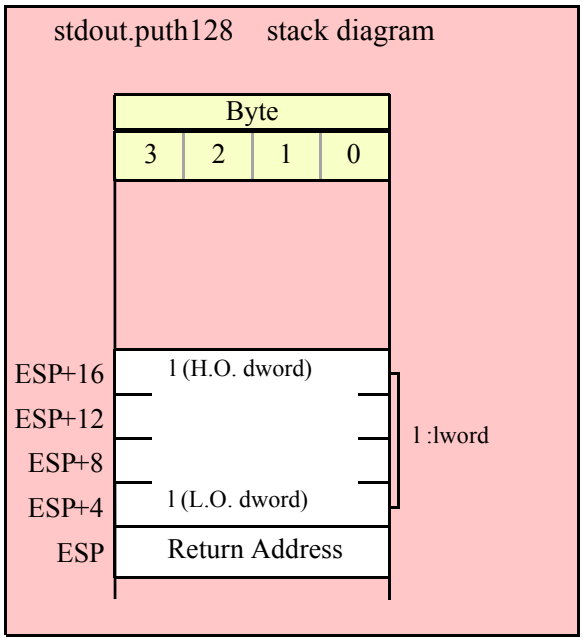
```
HLA high-level calling sequence examples:


stdout.putl( lwordVar );



HLA low-level calling sequence examples:

push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
call stdout.putl;
```



`stdout.puth128( l:lword );`

This procedure writes the value of l to the standard output using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

```
HLA high-level calling sequence examples:


stdout.puth128( lwordVar );



HLA low-level calling sequence examples:

push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
call stdout.puth128;
```

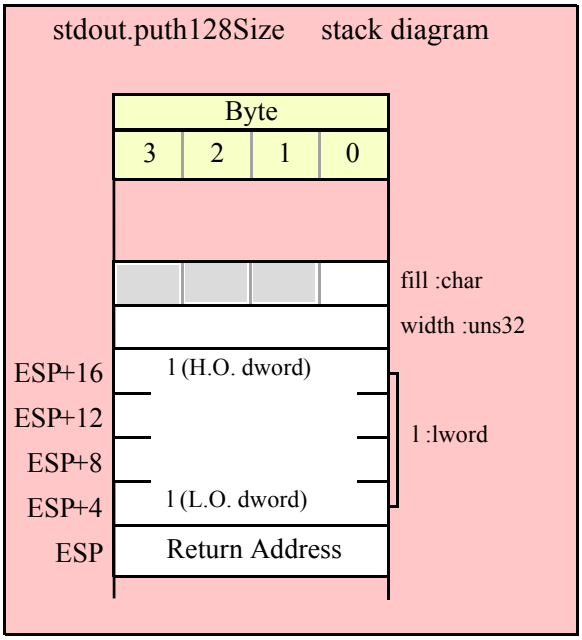


`stdout.puth128Size( l:lword; size:dword; fill:char );`

The stdout.puth128Size function writes an lword value to the standard output and it lets you specify a minimum field width and a fill character.

```
HLA high-level calling sequence examples:


stdout.puth128Size( tbyteVar, width, ' ' );



HLA low-level calling sequence examples:

push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
push( width );
pushd( ' ' );
call stdout.puth128Size;



// Assume fill char is in CH

push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.puth128Size;
```

```
    // Alternate method of the above

    push( (type dword lwordVar[12]));  // Push H.O. word first
    push( (type dword lwordVar[8]) );
    push( (type dword lwordVar[4]) );
    push( (type dword lwordVar));     // L.O. dword last
    push( width );
    sub( 4, esp );
    mov( ch, [esp] );
    call stdout.puth128Size;

    // If the fill char is a variable and
    // a register is available, try this code:

    push( (type dword lwordVar[12]));  // Push H.O. word first
    push( (type dword lwordVar[8]) );
    push( (type dword lwordVar[4]) );
    push( (type dword lwordVar));     // L.O. dword last
    push( width );
    movzx( fillChar, ebx ); // Assume EBX is available
    push( ebx );
    call stdout.puth128Size;


    // If the fill char is a variable and
    // no register is available, here's one
    // possibility:

    push( (type dword lwordVar[12]));  // Push H.O. word first
    push( (type dword lwordVar[8]) );
    push( (type dword lwordVar[4]) );
    push( (type dword lwordVar));      // L.O. dword last
    push( width );
    push( (type dword fillChar) );     // Chance of page crossing!
    call stdout.puth128Size;

    // In the very rare case that the above would
    // cause an illegal memory access, use this:

    push( (type dword lwordVar[12]));  // Push H.O. word first
    push( (type dword lwordVar[8]) );
    push( (type dword lwordVar[4]) );
    push( (type dword lwordVar));     // L.O. dword last
    push( width );
    sub( 4, esp );
    push( eax );
    movzx( fillChar, eax );
    mov( eax, [esp+4] );
    pop( eax );
    call stdout.puth128Size;
```



31.8 Signed Integer Output Routines

These routines convert signed integer values to string format and write that string to the standard output device. The stdout.putxxxSize functions contain width and fill parameters that let you specify the minimum field width when outputting a value.

If the absolute value of width is greater than the number of print positions the value requires, then these functions output width characters to the standard output device. If width is non-negative, then these functions right-justify the value in the output field; if value is negative, then these functions left-justify the value in the output field.

These functions print the fill character as the padding value for the extra print positions.

Note that unlike floating point values, these functions do not print a space in front of the value if it is non-negative.

xxxSize(value, width, fill);

Assuming "value" requires five print positions, "width" is eight, and fill is "f" then the xxxSize functions produce the string

f	f	f	V	A	L	U	E

Assuming "value" requires five print positions, "width" is minus eight, and fill is "f" then the xxxSize functions produce the string

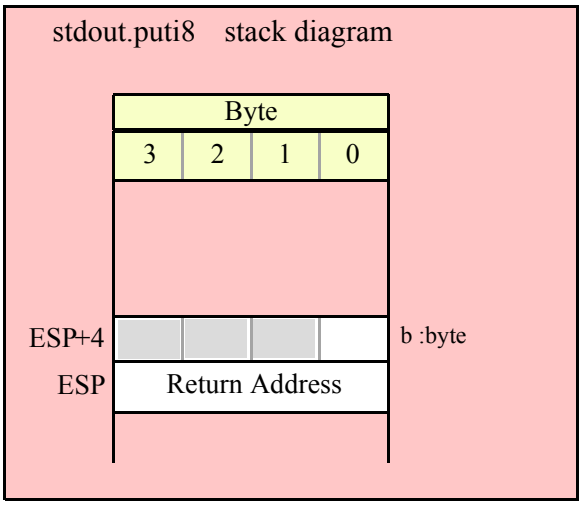
V	A	L	U	E	f	f	f

stdout.puti8 (b:byte);

This function converts the eight-bit signed integer you pass as a parameter to a string and writes this string to the standard output using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


stdout.puti8( byteVar );

// If the character is in a register (AL):
```

```
stdout.puti8( al );


HLA low-level calling sequence examples:


  // If "byteVar " is not one of the last three
  // bytes on a page of memory, you can do this:

push( (type dword byteVar ) );
call stdout.puti8;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call stdout.puti8;

// If no register is available, do something
// like the following code:

push( eax );
movzx( byteVar , eax );
push( eax );
call stdout.puti8;
pop( eax );

// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax );  // Assume byteVar is in AL
call stdout.puti8;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stdout.puti8;
```

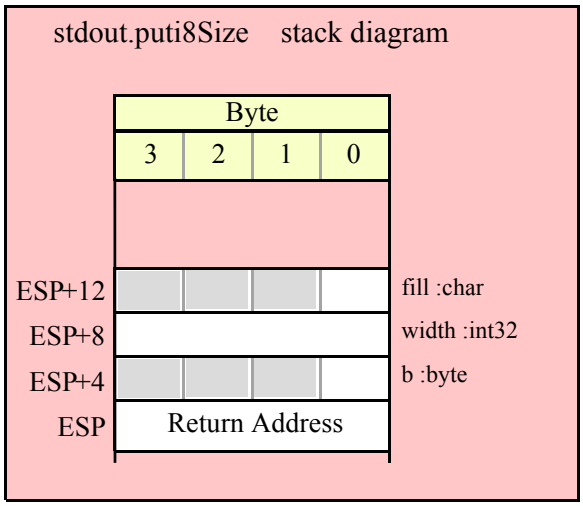


stdout.puti8Size (b:byte; width:int32; fill:char)

This function writes the eight-bit signed integer value you pass to the standard output using the width and fill values as specified above.

```
HLA high-level calling sequence examples:

stdout.puti8Size( byteVar, width, padChar );


HLA low-level calling sequence examples:


  // If "byteVar" and "padChar" are not one of the last three
  // bytes on a page of memory, you can do this:

push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call stdout.puti8Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( byteVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puti8Size;

// If no registers are available, do something
// like the following code:

push( eax );
movzx( byteVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
```

```
call stdout.puti8Size;
pop( eax );

// If "byteVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax );   // Assume byteVar is in AL
push( width );
push( ebx );   // Assume padChar is in BL
call stdout.puti8Size;

// Do the following if the characters are
// in AH, BH, CH, or DH:

xchg( al, ah );    // Assume byteVar is in AH
xchg( bl, bh );    // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call stdout.puti8Size;
```

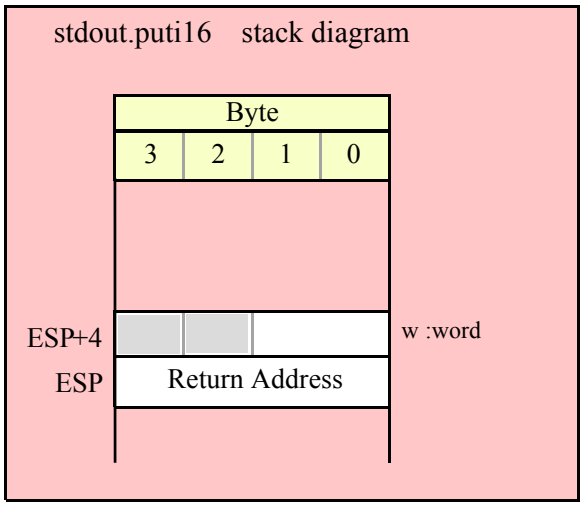


stdout.puti16(w:word);

This function converts the 16-bit signed integer you pass as a parameter to a string and writes this string to the standard output device using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


stdout.puti16( wordVar );

// If the word is in a register (AX):

stdout.puti16( ax );


HLA low-level calling sequence examples:
```

```
    // If "wordVar " is not one of the last three
    // bytes on a page of memory, you can do this:

push( (type dword wordVar) );
call stdout.puti16;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call stdout.puti16;

// If no register is available, do something
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
call stdout.puti16;
pop( eax );

// If the word value is in a 16-bit register
// then you can use code like the following:

push( eax );  // Assume wordVar is in AX
call stdout.puti16;
```



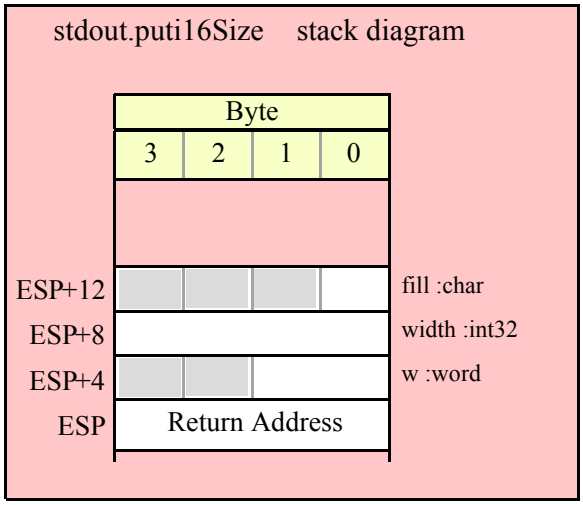
`stdout.puti16Size( w:word;  width:int32; fill:char );`

This function writes the 16-bit signed integer value you pass to the standard output using the width and fill values as specified above.

```
HLA high-level calling sequence examples:

stdout.puti16Size( wordVar, width, padChar );
```

HLA low-level calling sequence examples:

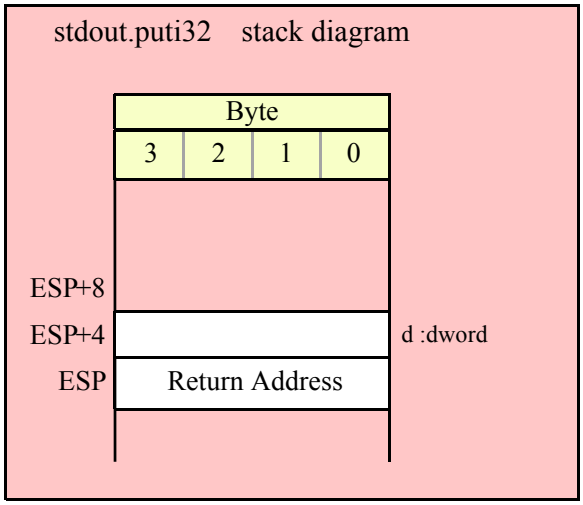
```
  // If "wordVar" and "padChar" are not one of the last three
  // bytes on a page of memory, you can do this:

push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call stdout.puti16Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( wordVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puti16Size;

// If no registers are available, do something
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call stdout.puti16Size;
pop( eax );

// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax );   // Assume wordVar is in AX
push( width );
push( ebx );   // Assume padChar is in BL
call stdout.puti16Size;
```



`stdout.puti32( d:dword );`

This function converts the 32-bit signed integer you pass as a parameter to a string and writes this string to the standard out using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


stdout.puti32( dwordVar );

// If the dword is in a register (EAX):

stdout.puti32( eax );


HLA low-level calling sequence examples:

push( dwordVar );
call stdout.puti32;

push( eax );
call stdout.puti32;
```



stdout.puti32Size(d:dword; width:int32; fill:char);

This function writes the 32-bit value you pass as a signed integer to the standard output device using the width and fill values as specified above.

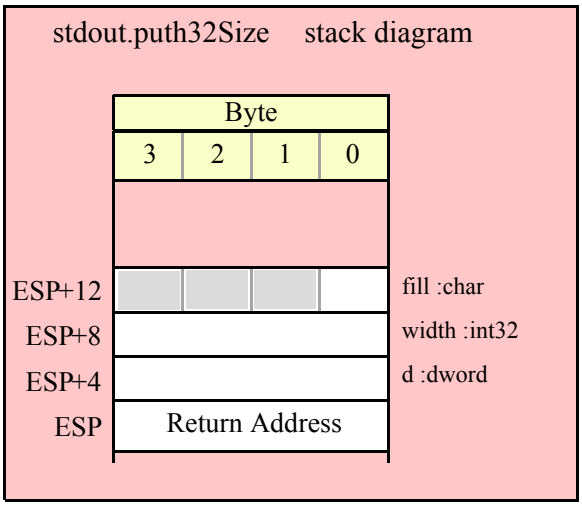
HLA high-level calling sequence examples:

```
stdout.putu32Size( dwordVar, width, ' ' );

// If the dword is in a register (EAX):

stdout.putu32Size( eax, width, cl );
```

HLA low-level calling sequence examples:

```
push( dwordVar );
push( width );
pushd( ' ' );
call stdout.putu32Size;

push( eax );
push( width );
push( ecx ); // fill char is in CL
call stdout.putu32Size;

// Assume fill char is in CH

push( eax );
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.putu32Size;

// Alternate method of the above

push( eax );
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.putu32Size;

// If the fill char is a variable and
// a register is available, try this code:

push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putu32Size;


// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( eax );
```

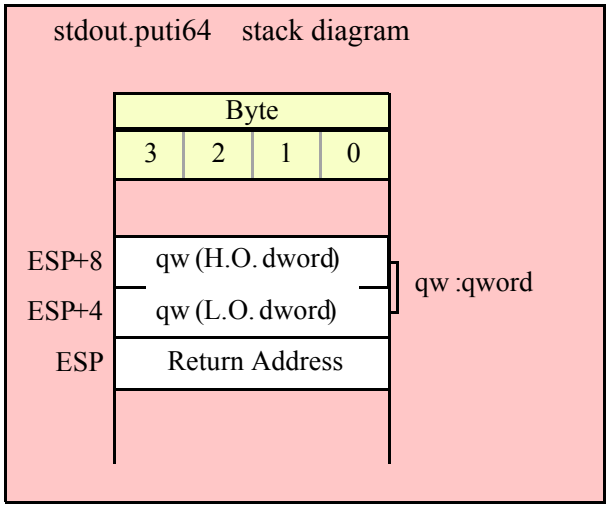
```
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stdout.putu32Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( eax );
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.puti32Size;
```



`stdout.puti64( q:qword );`

This function converts the 64-bit signed integer you pass as a parameter to a string and writes this string to the standard output using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


stdout.puti64( qwordVar );



HLA low-level calling sequence examples:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
call stdout.puti64;
```

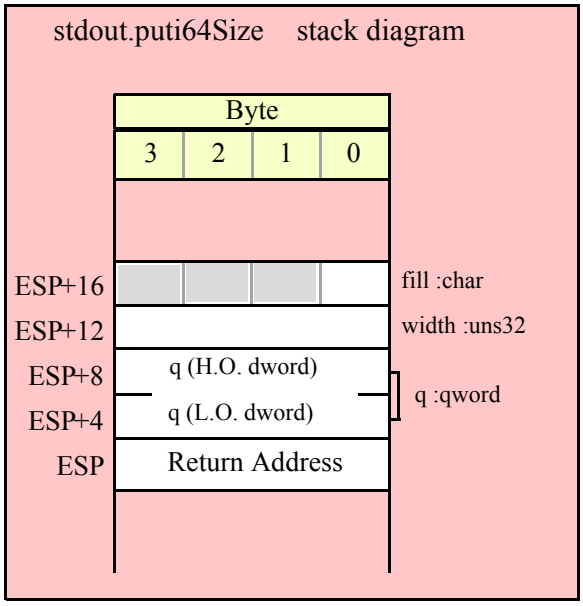


stdout.puti64Size(q:qword; width:int32; fill:char);

This function writes the 64-bit value you pass as a signed integer to the standard output using the width and fill values as specified above.

```
HLA high-level calling sequence examples:


stdout.puti64Size( qwordVar, width, ' ' );



HLA low-level calling sequence examples:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
pushd( ' ' );
call stdout.puti64Size;



push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call stdout.puti64Size;

// Assume fill char is in CH

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.puti64Size;

// Alternate method of the above

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
```

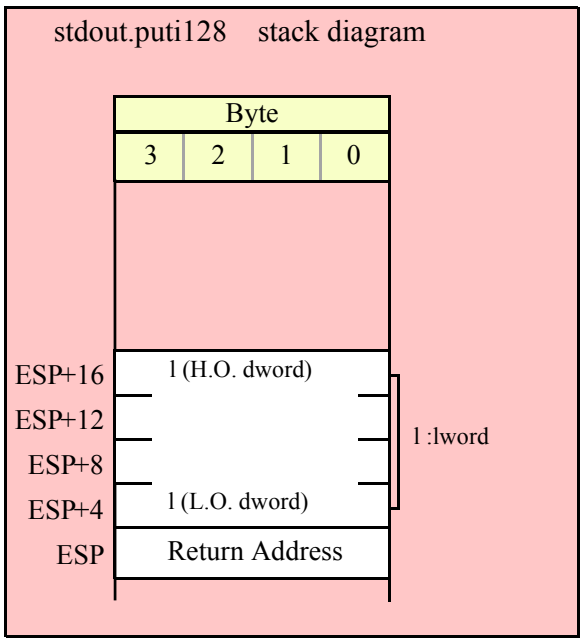
```
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.puti64Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puti64Size;


// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stdout.puti64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.puti64Size;
```



stdout.puti128(l:lword);

This function converts the 128-bit signed integer you pass as a parameter to a string and writes this string to the standard out using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


stdout.puti128( lwordVar );



HLA low-level calling sequence examples:

push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
call stdout.puti128;
```



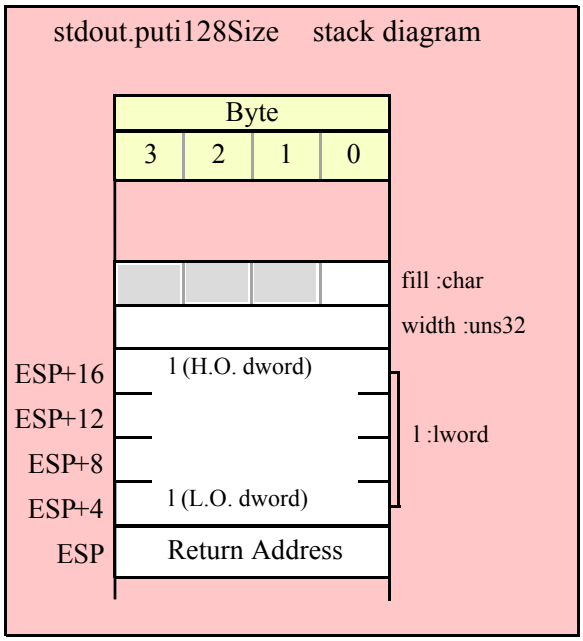
stdout.puti128Size(l:lword; width:int32; fill:char);

This function writes the 128-bit value you pass as a signed integer to the standard output device using the width and fill values as specified above.

```
HLA high-level calling sequence examples:


stdout.puti128Size( lwordVar, width, ' ' );



HLA low-level calling sequence examples:

push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
```

```
push( width );
pushd( ' ' );
call stdout.puti128Size;


// Assume fill char is in CH

push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.puti128Size;

// Alternate method of the above

push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.puti128Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puti128Size;


// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
push( (type dword fillChar) );     // Chance of page crossing!
call stdout.puti128Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
push( width );
```

```
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.puti128Size;
```



31.9 Unsigned Integer Output Routines

These routines convert unsigned integer values to string format and write that string to the standard output device. The stdout.putxxxSize functions contain width and fill parameters that let you specify the minimum field width when outputting a value.

If the absolute value of width is greater than the number of print positions the value requires, then these functions output width characters to the standard out. If width is non-negative, then these functions right-justify the value in the output field; if value is negative, then these functions left-justify the value in the output field.

These functions print the fill character as the padding value for the extra print positions.

xxxSize(value, width, fill);

Assuming "value" requires five print positions, "width" is eight, and fill is "f" then the xxxSize functions produce the string

f	f	f	V	A	L	U	E

Assuming "value" requires five print positions, "width" is minus eight, and fill is "f" then the xxxSize functions produce the string

V	A	L	U	E	f	f	f

stdout.putu8 (b:byte);

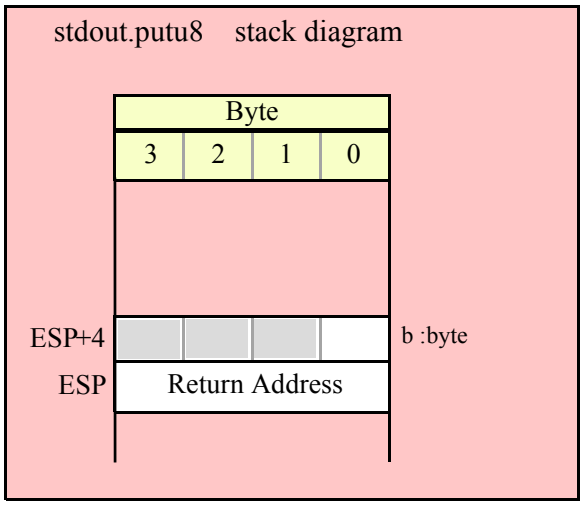
This function converts the eight-bit unsigned integer you pass as a parameter to a string and writes this string to the standard output device using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:
```

```
stdout.putu8( byteVar );

// If the character is in a register (AL):

stdout.putu8( al );
```

HLA low-level calling sequence examples:

```
  // If "byteVar " is not one of the last three
  // bytes on a page of memory, you can do this:

push( (type dword byteVar ) );
call stdout.putu8;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call stdout.putu8;

// If no register is available, do something
// like the following code:

push( eax );
movzx( byteVar , eax );
push( eax );
call stdout.putu8;
pop( eax );

// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax );  // Assume byteVar is in AL
call stdout.putu8;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stdout.putu8;
```

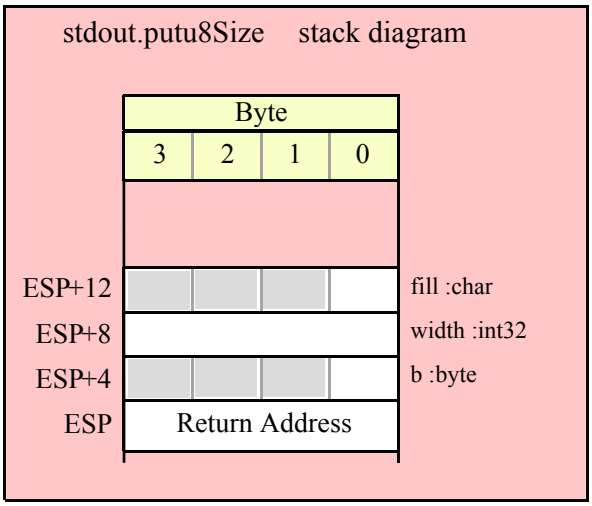


stdout.putu8Size(b:byte; width:int32; fill:char);

This function writes the unsigned eight-bit value you pass to the standard output using the width and fill values as specified above.

```
HLA high-level calling sequence examples:

stdout.putu8Size( byteVar, width, padChar );


HLA low-level calling sequence examples:


  // If "byteVar" and "padChar" are not one of the last three
  // bytes on a page of memory, you can do this:

push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call stdout.putu8Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( byteVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putu8Size;

// If no registers are available, do something
// like the following code:

push( eax );
movzx( byteVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
```

```
call stdout.putu8Size;
pop( eax );

// If "byteVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax );   // Assume byteVar is in AL
push( width );
push( ebx );   // Assume padChar is in BL
call stdout.putu8Size;

// Do the following if the characters are
// in AH, BH, CH, or DH:

xchg( al, ah );    // Assume byteVar is in AH
xchg( bl, bh );    // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call stdout.putu8Size;
```

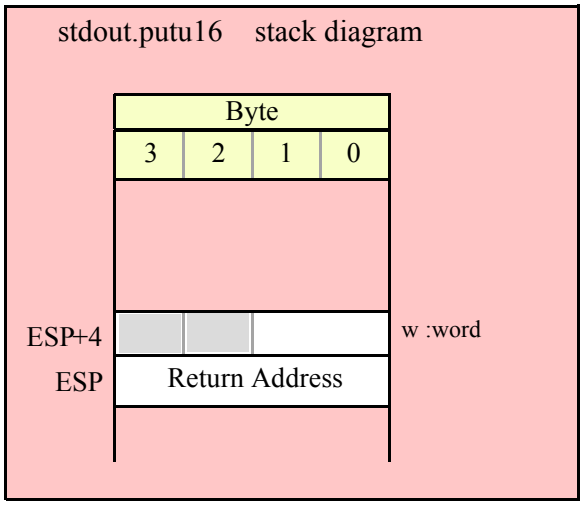


stdout.putu16(w:word);

This function converts the 16-bit unsigned integer you pass as a parameter to a string and writes this string to the standard output device using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


stdout.putu16( wordVar );

// If the word is in a register (AX):

stdout.putu16( ax );


HLA low-level calling sequence examples:
```

```
    // If "wordVar " is not one of the last three
    // bytes on a page of memory, you can do this:

  push( (type dword wordVar) );
  call stdout.putu16;

  // If you can't guarantee that the previous code
  // won't generate an illegal memory access, and a
  // 32-bit register is available, use code like
  // the following:

  movzx( wordVar, eax ); // Assume EAX is available
  push( eax );
  call stdout.putu16;

  // If no register is available, do something
  // like the following code:

  push( eax );
  movzx( wordVar, eax );
  push( eax );
  call stdout.putu16;
  pop( eax );

  // If the word value is in a 16-bit register
  // then you can use code like the following:

  push( eax );  // Assume wordVar is in AX
  call stdout.putu16;
```



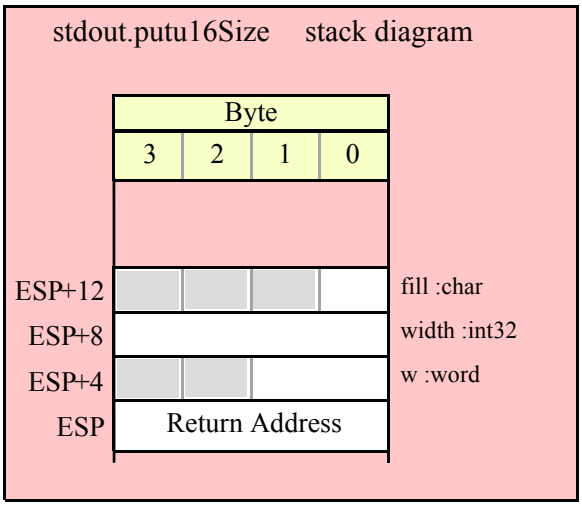
`stdout.putu16Size( w:word; width:int32; fill:char );`

This function writes the unsigned 16-bit value you pass to the standard out using the width and fill values as specified above.

```
  HLA high-level calling sequence examples:

  stdout.putu16Size( wordVar, width, padChar );
```

HLA low-level calling sequence examples:

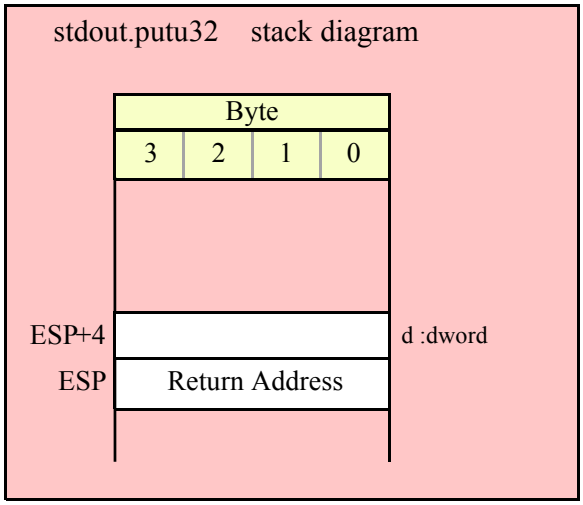
```
  // If "wordVar" and "padChar" are not one of the last three
  // bytes on a page of memory, you can do this:

push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call stdout.putu16Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( wordVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putu16Size;

// If no registers are available, do something
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call stdout.putu16Size;
pop( eax );

// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax );   // Assume wordVar is in AX
push( width );
push( ebx );   // Assume padChar is in BL
call stdout.putu16Size;
```



`stdout.putu32( d:dword );`

This function converts the 32-bit unsigned integer you pass as a parameter to a string and writes this string to the standard out using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


stdout.putu32( dwordVar );

// If the dword is in a register (EAX):

stdout.putu32( eax );


HLA low-level calling sequence examples:

push( dwordVar );
call stdout.putu32;

push( eax );
call stdout.putu32;
```

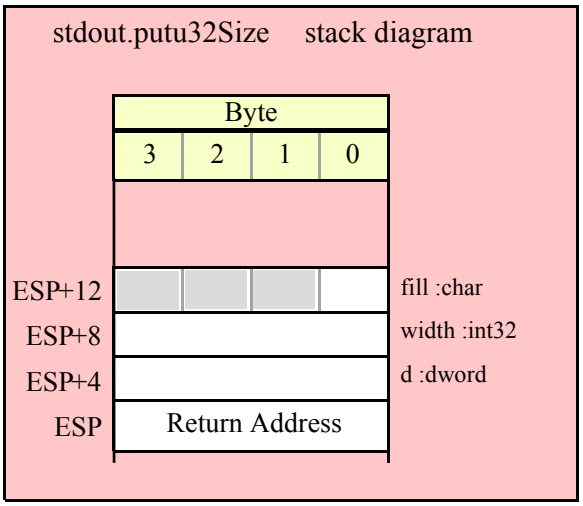


stdout.putu32Size(d:dword; width:int32; fill:char);

This function writes the unsigned 32-bit value you pass to the standard out using the width and fill values as specified above.

```
HLA high-level calling sequence examples:


stdout.putu32Size( dwordVar, width, ' ' );

// If the dword is in a register (EAX):

stdout.putu32Size( eax, width, cl );



HLA low-level calling sequence examples:

push( dwordVar );
push( width );
pushd( ' ' );
call stdout.putu32Size;

push( eax );
push( width );
push( ecx ); // fill char is in CL
call stdout.putu32Size;

// Assume fill char is in CH

push( eax );
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.putu32Size;

// Alternate method of the above

push( eax );
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.putu32Size;

// If the fill char is a variable and
// a register is available, try this code:

push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putu32Size;


// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( eax );
```

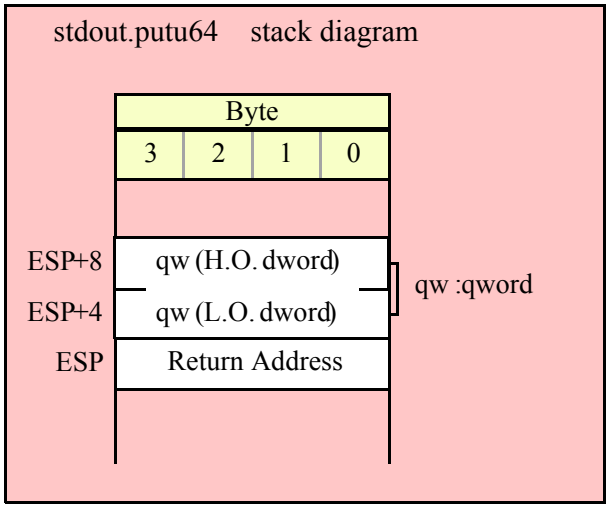
```
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stdout.putu32Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( eax );
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.putu32Size;
```



`stdout.putu64( q:qword );`

This function converts the 64-bit unsigned integer you pass as a parameter to a string and writes this string to the standard output device using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


stdout.putu64( qwordVar );



HLA low-level calling sequence examples:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
call stdout.putu64;
```

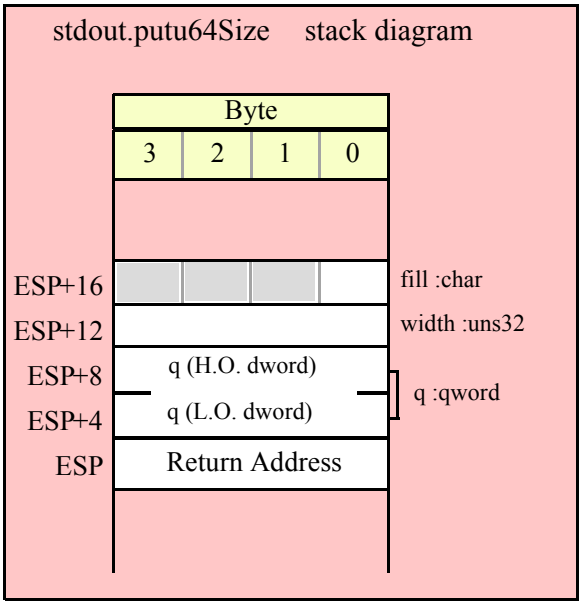


stdout.putu64Size(q:qword; width:int32; fill:char);

This function writes the unsigned 64-bit value you pass to the output using the width and fill values as specified above.

```
HLA high-level calling sequence examples:


stdout.putu64Size( qwordVar, width, ' ' );



HLA low-level calling sequence examples:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
pushd( ' ' );
call stdout.putu64Size;



push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call stdout.putu64Size;

// Assume fill char is in CH

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.putu64Size;

// Alternate method of the above

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
```

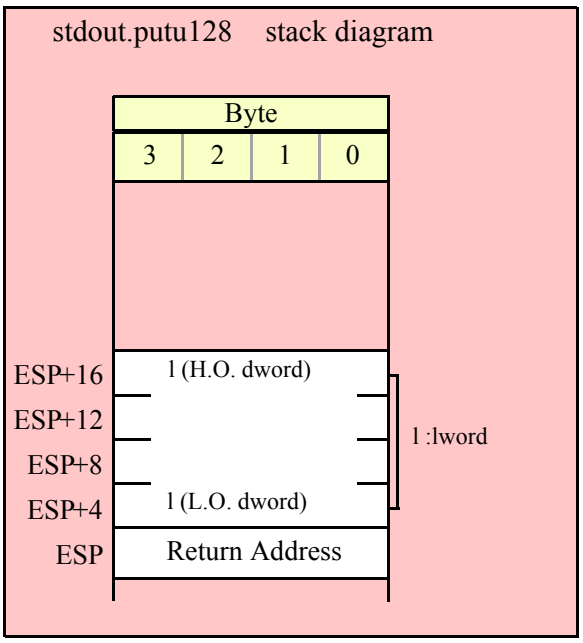
```
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.putu64Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putu64Size;


// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stdout.putu64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.putu64Size;
```



stdout.putu128(l:lword);

This function converts the 128-bit unsigned integer you pass as a parameter to a string and writes this string to the standard out using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


stdout.putu128( lwordVar );



HLA low-level calling sequence examples:

push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
call stdout.putu128;
```



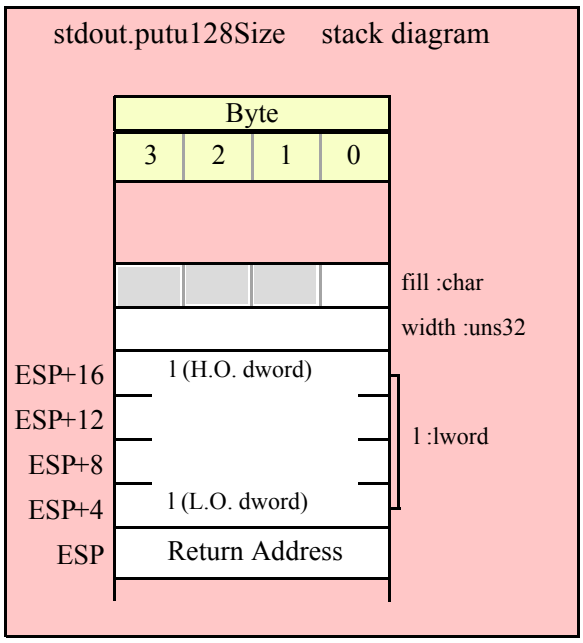
stdout.putu128Size(l:lword; width:int32; fill:char)

This function writes the unsigned 128-bit value you pass to the standard out using the width and fill values as specified above.

```
HLA high-level calling sequence examples:


stdout.putu128Size( lwordVar, width, ' ' );



HLA low-level calling sequence examples:

push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
```

```
push( width );
pushd( ' ' );
call stdout.putu128Size;


// Assume fill char is in CH

push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.putu128Size;

// Alternate method of the above

push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.putu128Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putu128Size;


// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
push( (type dword fillChar) );     // Chance of page crossing!
call stdout.putu128Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
push( width );
```

```
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.putu128Size;
```



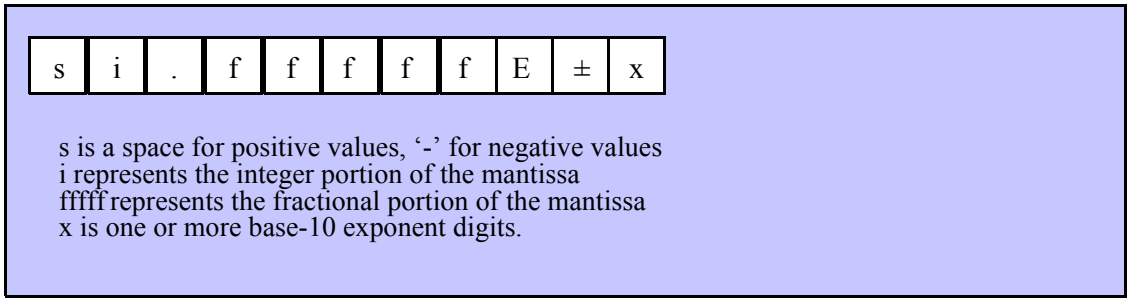
31.10 Floating Point Output Routines

The HLA standard output module provides several procedures you can use to write floating point values to the standard output device. The following subsections describe these routines.

31.10.1 Real Output Using Scientific Notation

The floating point numeric output routines translate the three different binary floating point formats to their string representation and then write this string to the standard output. There are two generic classes of these routines: those that convert their values to exponential/scientific notation and those that convert their string to a decimal form.

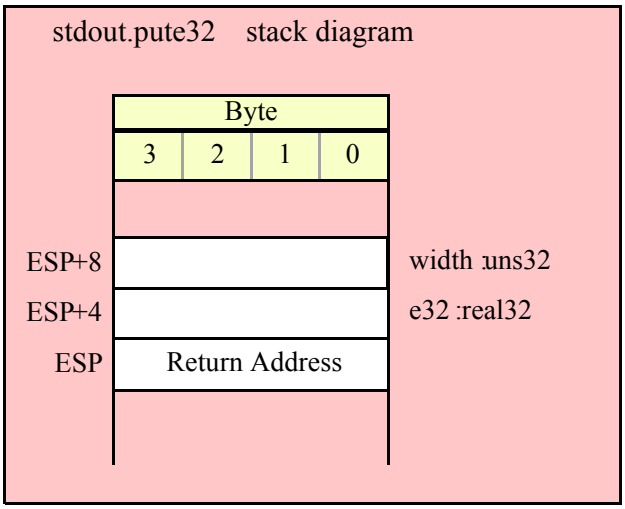
The stdout.pute80, stdout.pute64, and stdout.pute32 routines convert their values to a string using scientific notation. These three routines each have two parameters: the value to output and the field width of the result. These routines produce a string with the following format:



stdout.pute32(r:real32; width:uns32);

This function writes the 32-bit single precision floating point value passed in r to the standard out using scientific/exponential notation. This procedure prints the value using width print positions in the output. width should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 32-bit extended precision floating point values support about 6-7 significant digits. So a width value that yeilds more than seven mantissa digits will produce garbage output in the low order digits of the number.

```
HLA high-level calling sequence examples:


stdout.pute32( r32Var, width );

// If the real32 value is in an FPU register (ST0):

var
  r32Temp:real32;
  .
  .
  .
fstp( r32Temp );
stdout.pute32( r32Temp, 12 );



HLA low-level calling sequence examples:

push( (type dword r32Var) );
push( width );
call stdout.pute32;

sub( 4, esp );
fstp( (type real32 [esp]) );
pushd( 12 );
call stdout.pute32;
```

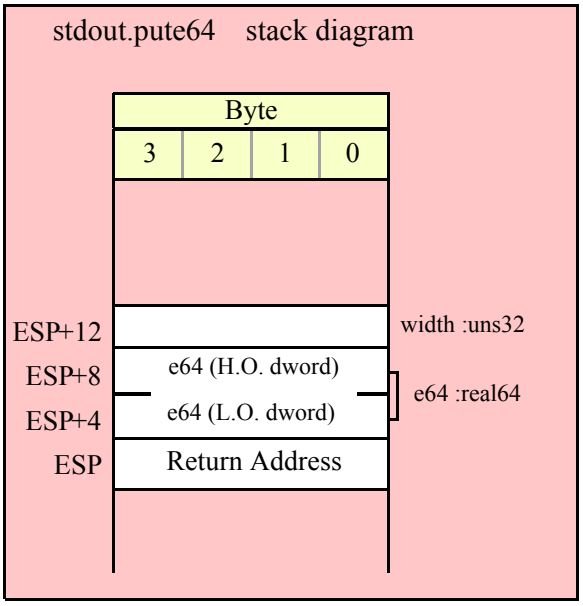


stdout.pute64(r:real64; width:uns32);

This function writes the 64-bit double precision floating point value passed in r to the standard output using scientific/exponential notation. This procedure prints the value using width print positions in the output. width should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 64-bit double precision floating point values support about 15 significant digits. So a

width value that yeilds more than 15 mantissa digits will produce garbage output in the low order digits of the number.

```
HLA high-level calling sequence examples:


stdout.pute64( r64Var, width );

// If the real64 value is in an FPU register (ST0):

var
  r64Temp:real64;
  .
  .
  .
fstp( r64Temp );
stdout.pute64( r64Temp, 12 );


HLA low-level calling sequence examples:

push( (type dword r64Var[4]));
push( (type dword r64Var[0]));
push( width );
call stdout.pute64;

sub( 8, esp );
fstp( (type real64 [esp]) );
pushd( 12 );
call stdout.pute64;
```

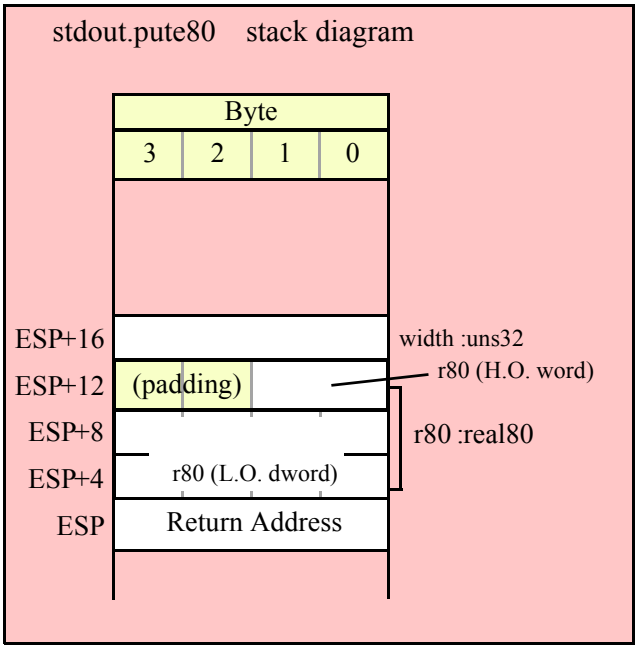


`stdout.pute80( r:real80; width:uns32 );`

This function writes the 80-bit extended precision floating point value passed in r to the standard output device using scientific/exponential notation. This procedure prints the value using width print positions in the standard out. width should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 80-bit extended precision floating point values support

about 18 significant digits. So a width value that yeilds more than 18 mantissa digits will produce garbage output in the low order digits of the number.

```
HLA high-level calling sequence examples:


stdout.pute80( r80Var, width );

// If the real80 value is in an FPU register (ST0):

var
  r80Temp:real80;
  .
  .
  .
fstp( r80Temp );
stdout.pute80( r80Temp, 12 );



HLA low-level calling sequence examples:

pushw( 0 ); // A word of padding.
push( (type word r80Var[8]));
push( (type dword r80Var[4]));
push( (type dword r80Var[0]));
push( width );
call stdout.pute80;

sub( 12, esp );
fstp( (type real80 [esp]) );
pushd( 12 );
call stdout.pute80;
```

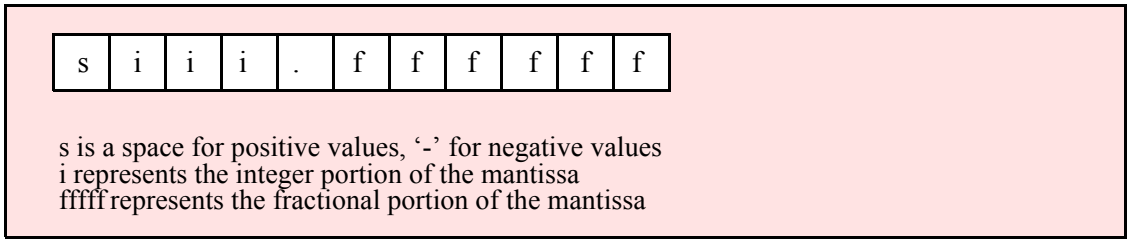


31.10.2 Real Output Using Decimal Notation

Although scientific (exponential) notation is the most general display format for real numbers, real numbers you display in this format are very difficult to read. Therefore, the HLA stdout module also provides a set of

functions that output real values using the decimal representation. Although you cannot (practically) use these decimal output routines for all real values, they are applicable to a wide variety of common numbers you will use in your programs.

These functions require four parameters: the real value to convert, the width of the converted value, the number of digit positions to the right of the decimal point, and a padding character. These functions write their values using the following string format:

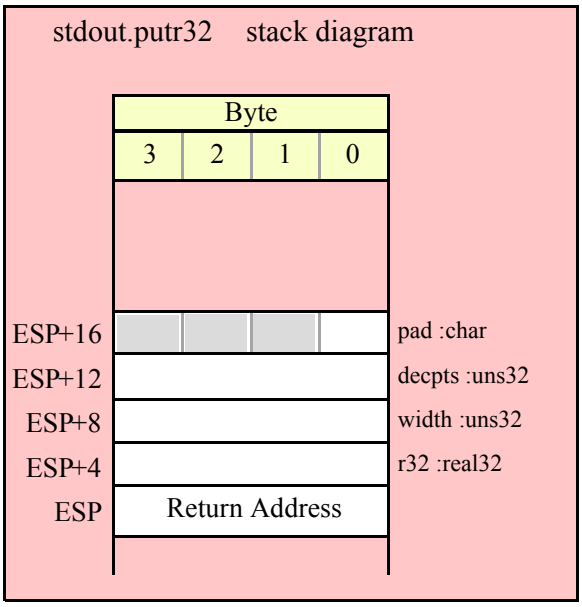


stdout.putr32(r:real32; width:uns32; decpts:uns32; pad:char);

This procedure writes a 32-bit single precision floating point value to the standard output as a string. The string consumes exactly width characters in the standard output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than width characters, then this procedure uses the value of pad as the padding character to fill the output with width characters. The number is right-justified within the output field (that is, this function prints the padding characters before the string representation of the number).

```
HLA high-level calling sequence examples:


stdout.putr32( r32Var, width, decpts, fill );
stdout.putr32( r32Var, 10, 2, '*' );

// If the real32 value is in an FPU register (ST0):

var
  r32Temp   :real32;
  .
  .
  .
fstp( r32Temp );
stdout.putr32( r32Temp, 12, 2, al );



HLA low-level calling sequence examples:

push( (type dword r32Var) );
push( width );
push( decpts );
movzx( fill, eax );
push( eax );
call stdout.putr32;

push( (type dword r32Var) );
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call stdout.putr32;

sub( 4, esp );
fstp( (type real32 [esp]) );
pushd( 12 );
```

```
sub( 4, esp );
push( eax );
movzx( fill, eax ):   // If memory fault were possible
mov( eax, [esp+4] );  // in above code.
pop( eax );
call stdout.putr32;
```

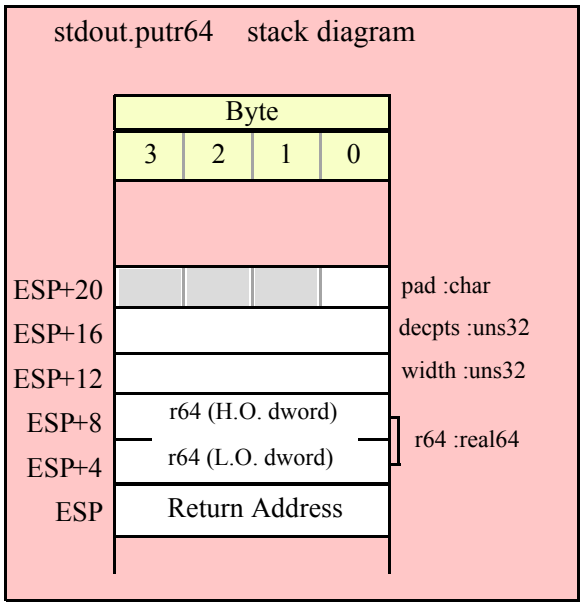


stdout.putr64(r:real64; width:uns32; decpts:uns32; pad:char);

This procedure writes a 64-bit double precision floating point value to the standard output device as a string. The string consumes exactly width characters in the output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than width characters, then this procedure uses the value of pad as the padding character to fill the output with width characters.

```
HLA high-level calling sequence examples:


stdout.putr64( r64Var, width, decpts, fill );
stdout.putr64( r64Var, 10, 2, '*' );

// If the real64 value is in an FPU register (ST0):

var
  r64Temp:real64;
  .
  .
  .
fstp( r64Temp );
stdout.putr64( r64Temp, 12, 2, al );



HLA low-level calling sequence examples:

push( (type dword r64Var[4]) );
push( (type dword r64Var) );
push( width );
push( decpts );
```

```
movzx( fill, eax );
push( eax );
call stdout.putr64;

push( (type dword r64Var[4]) );
push( (type dword r64Var) );
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call stdout.putr64;

sub( 8, esp );
fstp( (type real64 [esp]) );
pushd( 12 );
sub( 4, esp );
push( eax );
movzx( fill, eax ):// If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
call stdout.putr64;
```

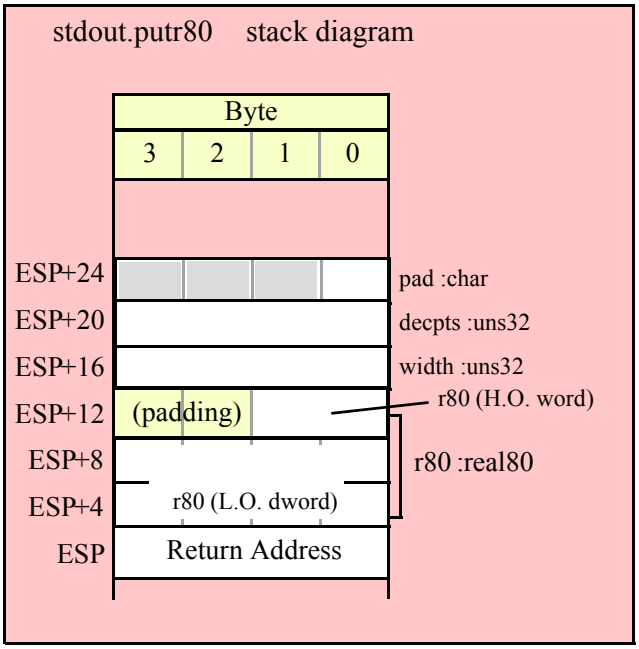


stdout.putr80(r:real80; width:uns32; decpts:uns32; pad:char);

This procedure writes an 80-bit extended precision floating point value to the output as a string. The string consumes exactly width characters in the output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than width characters, then this procedure uses the value of pad as the padding character to fill the output with width characters.

```
HLA high-level calling sequence examples:


stdout.putr80( r80Var, width, decpts, fill );
stdout.putr80( r80Var, 10, 2, '*' );

// If the real80 value is in an FPU register (ST0):

var
  r80Temp:real80;
```

```
    .
    .
    .
fstp( r80Temp );
stdout.putr80( r80Temp, 12, 2, al );


HLA low-level calling sequence examples:

pushw( 0 );
push( (type word r80Var[8]) );
push( (type dword r80Var[4]) );
push( (type dword r80Var) );
push( width );
push( decpts );
movzx( fill, eax );
push( eax );
call stdout.putr80;

pushw( 0 );
push( (type word r80Var[8]) );
push( (type dword r80Var[4]) );
push( (type dword r80Var) );
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call stdout.putr80;

sub( 12, esp );
fstp( (type real80 [esp]) );
pushd( 12 );
sub( 4, esp );
push( eax );
movzx( fill, eax ):// If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
call stdout.putr80;
```



31.11 Generic Standard Output Routine

stdout.put(list_of_items);

stdout.put is a macro that automatically invokes an appropriate stdout output routine based on the type of the parameter(s) you pass it. This is a very convenient output routine and is probably the stdout output call you will use most often in your programs. Keep in mind that this macro is not a single function call; instead, HLA translates this macro into a sequence of calls to procedures like stdout.putu32, stdout.puts, etc.

stdout.put is a macro that provides a flexible syntax for outputting data to the standard output device. This macro allows a variable number of parameters. For each parameter present in the list, stdout.put will call the appropriate routine to emit that data, according to the type of the parameter. Parameters may be constants, registers, or memory locations. You must separate each macro parameter with a comma.

Here is an example of a typical invocation of stdout.put:

```
stdout.put( "I=", i, " j=", j, nl );

The above is roughly equivalent to

stdout.puts( "I=" );
stdout.putu32( i );
stdout.puts( " j=" );
stdout.putu32( j );
stdout.newln();
```

This assumes, of course, that i and j are int32 variables.

The stdout.put macro also lets you specify the minimum field width for each parameter you specify. To print a value using a minimum field width, follow the object you wish to print with a colon and the value of the minimum field width. The previous example, using field widths, could look like the following:

```
stdout.put( "I=", i:2, " j=", j:5, nl );
```

Although this example used the literal decimal constants two and five for the field widths, keep in mind that register values and memory value (integers, anyway) are prefectly legal here.

For floating point numbers you wish to display in decimal form, you can specify both the minimum field width and the number of digits to print to the right of the decimal point by using the following syntax:

```
stdout.put( "Real value is ", f:10:3, nl );
```

The stdout.put macro can handle all the basic primitive types, including boolean, unsigned (8, 16, 32, 64, 128), signed (8, 16, 32, 64, 128), character, character set, real (32, 64, 80), string, and hexadecimal (byte, word, dword, qword, lword).

There is a known "design flaw" in the stdout.put macro. You cannot use it to print HLA intermediate variables (i.e., non-local VAR objects). The problem is that HLA's syntax for non-local accesses takes the form "reg32:varname" and stdout.put cannot determine if you want to print reg32 using varname print positions versus simply printing the non-local varname object. If you want to display non-local variables you must copy the non-local object into a register, a static variable, or a local variable prior to using stdout.put to print it. Of course, there is no problem using the other stdout.put*XXXX* functions to display non-local VAR objects, so you can use those as well.