8 Conversions (conv.hhf)

This unit contains routines that perform general conversions from one data type to another. Primarily, this unit supplies the routines that convert various data types to and from string form.

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

Most string conversion routines take two forms: one version that writes data to an existing (preallocated) string and one that automatically allocates storage for a new string on the heap. Those functions that automatically allocate storage generally have a name that begins with "a_" (for allocate) whereas the functions that use a preallocated string do not have such a prefix. For example, the conv.h8ToStr function converts an 8-bit integer to a string using hexadecimal representation and stores the result in a preallocated string object. The conv.a_h8ToStr function converts an 8-bit value to a (hexadecimal) string that it allocates on the heap; conv.a_h8ToStr returns a pointer to that string in the EAX register.

An important point to keep in mind is that string variables are pointers. Unless you call a function that allocates storage for a string (i.e., one of the "a_..." functions), you must ensure that you've allocated sufficient storage to hold any string result the function produces. Failure to do so will produce a memory access error, null pointer reference error, or string overflow error. Remember, simply declaring a string variable does not automatically allocate storage for any string data; the declaration only allocates storage for the string pointer. You must call a function such as str.alloc to actually allocate the string data.

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 the FPU: The Standard Library code makes occasional use of the FPU, particularly when converting between real and string formats and when computing 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).

8.1 Buffer vs. String Conversions

The Standard Library supports two generic types of numeric-to-string conversions – output to a string variable (an "HLA string" object) and output to a memory buffer. The string conversion routines are the safest to use, but the buffer conversion routines are a bit more general.

If you're working with HLA-style string objects, then using the conversion-to-string functions make the most sense because you get to take full advantage of range checking and other facilities that are possible with the string format. Furthermore, you can use the Standard Library string manipulation functions to process such strings once the conversion is complete.

There are two drawbacks to the string conversion routines (versus the buffer conversion routines):

- You may intend to pass the converted data on to some other routine that doesn't know anything about the HLA string format, so you may need to produce the string using a different data structure.
- If you want to produce a longer string as a sequence of conversion operations, it is slightly more efficient to do the conversion to a single buffer (which may very well be an HLA string object) and fix up the string data structure afterwards.

Perhaps the most common example of a non-HLA-string data type you'll encounter is the simple zero-terminated string (the word "simple" appears here because HLA strings are zero-terminated and you can often use them whenever you need a zero-terminated string). Consider the conv.i32ToBuf routine that converts a 32-bit signed integer to the corresponding sequence of characters. This function stores that characters at the memory address passed in EDI and upon return EDI points at the first byte beyond the converted sequence, e.g.,

```
// Stores the characters "12345" at [edi]

conv.i32ToBuf( 12345, [edi] );
```

Upon return from this function, EDI will contain a value that is 5 greater than it was upon entry, and the five memory locations that EDI has skipped over will contain the characters "12345". Note that this string is not zero-terminated, but you can easily zero-terminate it by storing a zero byte at the location where EDI points upon return from the function:

```
// Stores the characters "12345" at [edi] and zero
// terminates the string.

conv.i32ToBuf( 12345, [edi] );
mov( 0, (type byte [edi]));
```

As a final example, suppose you want to build up an HLA style string by concatenating two converted strings together. You could do something like the following:

```
// Produces " 12345  67890" in fullStr

conv.i32ToStr( 12345, 6, ' ', leftStr );
conv.i32ToStr( 67890, 7, ' ', rightStr
str.cat( leftStr, rightStr, fullStr );
```

The only problem with this approach is that there is unnecessary string processing (e.g., data copying) taking place. If efficiency is paramount, and you don't need the intermediate conversions (leftStr and rightStr), then you can do this sequence a little bit faster by generating the two strings in place as follows:

```
mov( fullStr, edi );            // Point EDI at start of string data
mov( edi, ebx );                // Save to compute length
conv.i32ToBuf( 12345, 6, ' ' ); // Store " 12345" at fullStr
conv.i32ToBuf( 67890, 7, ' ' ); // Store "  67890" at fullStr+6
mov( 0, (type byte [edi]));     // HLA strings must be zero terminated
sub( fullStr, edi );            // Compute string length
mov( edi, (type str.strRec [ebx]).length ); // Save new length.
```

As the number of objects you append to the string increases, this scheme becomes even more efficient than using the str.cat approach. The code above, of course, assumes that you've already allocated a sufficient amount of string storage for the leftStr, rightStr, and fullStr string variables.

8.2 Conversion Format Control

The following functions control the numeric conversion process.

8.2.1 Underscore Control

When converting numeric data types to strings, the standard library offers the option of inserting underscores at appropriate places in the numbers (i.e., where you would normally expect a digit separator to

appear, such as a comma [U.S.] or period [Europe]). The conv.setUnderscores and conv.getUnderscores functions control the operation of this feature.

The standard library conversion functions will inject underscores into hexadecimal, unsigned integers, and signed integers if the feature is enabled. For hexadecimal output the standard library conversion routines will emit an underscore between every fourth and fifth digit, starting with the L.O. digit (e.g., 1234_5678). For decimal integers (signed or unsigned), the conversion routines emit an underscore between each third and fourth digit starting with the L.O. digit (e.g., 123_456_789).

Note that the conversion routines do *not* emit underscores into conversions of floating-point/real values.

Thread Issues: Because the standard library maintains the internal underscore flag as a static object there will be some problems if you attempt to read and set the underscore flag in multiple threads running in the same address space. In particular, if you read the underscores flag and save it, set it to a different value, do some conversions, and then restore the underscores flag its original value, it is quite possible that another thread could do some conversions between those two points and produce incorrect output. Indeed, it would even be possible for half the number to contain underscores and the other half not contain underscores, depending on where the system interrupts the second thread. The current library code does not address this issue because the cost is very high to solve a problem that almost never occurs (most assembly applications are single-threaded). However, if you are writing a multi-threaded application, you should note that constantly changing the underscores flag is not a good idea – you should try to set the flag once, at the beginning of your program, and leave it alone throughout the program's execution. If you must change the underscore flag setting on a regular basis within a multi-threaded application, you should put appropriate locks around all calls to conversion routines (and those routines, such as the I/O routines, that call the conversion code) to protect the settings.

Current plans are to make the Standard Library thread-safe when the threads module is added to the library.

`conv.setUnderscores( onOff: boolean );`

The conv.setUnderscores function lets you enable or disable the emission of underscores in numeric values. Passing true enables underscore emission, passing false disables it.

For efficiency reasons, the standard library routines always pass all parameters as a multiple of four bytes. The *onOff* Boolean parameter consumes the L.O. byte of the double word passed on the stack. The conv.setUnderscore routine ignores the H.O. three bytes of the value passed for this parameter, though by convention (to make debugging a little easier) you should try to pass zeros in the H.O. three bytes if it is not inconvenient to do so.

When passing a Boolean constant, you should simply push the dword containing the value true (1) or false (0), e.g,

```
pushd( true );
call conv.setUnderscores;
   .
   .
   .
pushd( false );
call conv.setUnderscores;
```

When passing the Boolean value in one of the 8-bit registers AL, BL, CL or DL, you should simply push the 32-bit register that holds the 8-bit register, e.g.,

```
push( eax );  // Pushes AL onto the stack
call conv.setUnderscores;
push( ebx );  // Pushes BL onto the stack
call conv.setUnderscores;
```

Note that this trick does not apply to the AH, BH, CH, or DH registers. The best code to use when you need to push these registers is to drop the stack down by four bytes and then move the desired register into the memory location you've just created on the stack, e.g.,

```
sub( 4, esp );
mov( AH, [esp] ); // Pushes AH onto the stack
call conv.setUnderscores;
   .
```

```
    .
    .
sub( 4, esp );
mov( BH, [esp] ); // Pushes BH onto the stack
call conv.setUnderscores;
```

Here's another way you can accomplish this (a little slower, but leaves zeros in the H.O. three bytes):

```
pushd( 0 );
mov( CH, [esp] ); // Pushes CH onto the stack
call conv.setUnderscores;
```

When passing a Boolean variable, you should try to push the Boolean value and the following three bytes, using code like the following (HLA syntax):

```
pushd( (type dword boolVar ));
call conv.setUnderscores;
```

There is one drawback to the approach above. In three very rare cases the code above could cause a segmentation fault. If the Boolean variable is located on the last three bytes of a page in memory (4,096 bytes) and the next memory page is not readable, the system will generate a fault if you attempt to push all four bytes. In such a case, the next best solution, if a register is available, is to move the Boolean value into AL, BL, CL, or DL and push the corresponding 32-bit register. If no registers are available, then you can write code like the following:

```
push( eax );
push( eax );
mov( boolVar, al );
mov( al, [esp+4] );
pop( eax );
call conv.setUnderscores;
```

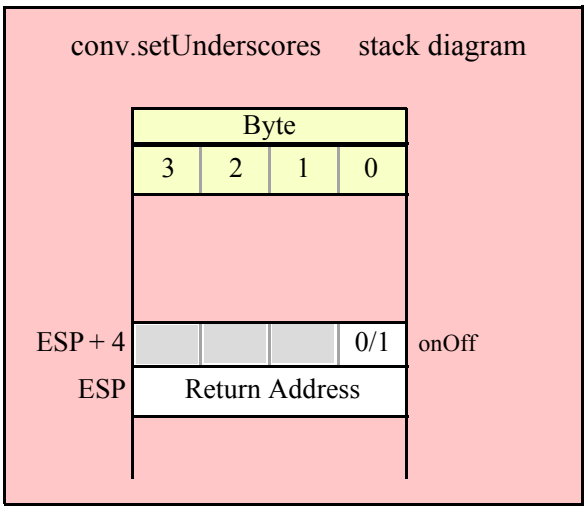
This code is ugly and slightly inefficient, but it will always work (assuming, of course, you don't get a stack overflow).

The HLA compiler will generate code similar to this last example if you pass a boolean variable as the actual parameter to conv.setUnderscores:

```
conv.setUnderscores( boolVar );
```

Therefore, if efficiency is a concern to you, you should try to load the Boolean variable (boolVar in this example) into AL, BL, CL, or DL prior to calling conv.setUnderscores, e.g.,

```
mov( boolVar, al );
conv.setUnderscores( al );
```




```
conv.getUnderscores;  @returns( "eax" );
```

You can test the current state of the underscore conversion by calling conv.getUnderscores. This function call returns the boolean result in EAX (true means underscores will be output); AL will contain the actual Boolean value and the H.O. three bytes of EAX will all contain zero. This routine does not have any parameters.

The following example demonstrates how to preserve the value of the internal underscores flag across some section of code:

```
conv.getUnderscores();
mov( al, saveUnderscores );
conv.setUnderscores( true );
  .
  .
  .
mov( saveUnderscores, al );
conv.setUnderscores( al );
```

Note: Do not try to access the internal underscores flag directly in your program. Always use the conv.setUnderscores and conv.getUnderscores accessor functions. In a future version of the Standard Library, the internal representation of this flag *will* change and any code that accesses it directly will break at that point. However, if you call conv.setUnderscores and conv.getUnderscores, you're guaranteed that the internal implementation will be hidden from you and your code will not fail when the internal representation changes.

8.2.2 Delimiter Control

During the conversion from string to a numeric form, the conversion routines will skip over zero or more delimiter characters and then process all numeric digits (including hexadecimal digits, if doing a hexadecimal conversion) up to the end of string or the first delimiter character it finds. If a conversion function encounters a value that is not a valid digit or delimiter character, it will raise a conversion exception or an illegal character exception By default, the delimiter characters are members of the following set:

```
Delimiters: cset :=
    {
        #0,    // End of string
        #9,    // Tab
```

```
            #10,   // Line feed
            #13,   // Carriage return
            ' ',   // Space
            ',',   // Comma
            ';',   // Semicolon
            ':'    // Colon
        };
```

While this default delimiters character set is probably appropriate for most applications, some programmers may want to add or remove characters from this set based on their application requirements. The standard library provides two routines that provide access to this internal character set object: conv.getDelimiters and conv.setDelimiters. You should always use these routines to access this character set object rather than accessing it directly (as an external object).

Thread Issues: Because the standard library maintains the internal delimiters character set as a static object there will be some problems if you attempt to read and set the delimiters in multiple threads running in the same address space. In particular, if you read the delimiters character set and save it, set it to a different value, do some conversions, and then restore the delimiters to the original value, it is quite possible that another thread could do some conversions between those two points and produce incorrect. The current library code does not address this issue because the cost is very high to solve a problem that almost never occurs (most assembly applications are single-threaded). However, if you are writing a multi-threaded application, you should note that constantly changing the delimiters character set is not a good idea – you should try to set the delimiters once, at the beginning of your program, and leave them alone throughout the program's execution. If you must change the delimiters character set on a regular basis within a multi-threaded application, you should put appropriate locks around all calls to conversion routines (and those routines, such as the I/O routines, that call the conversion code) to protect the settings.

Current plans are to make the delimiters character set object thread-safe when the processes module is added to the library.

Note: Do not try to access the internal delimiters character set directly in your program. Always use the conv.setDelimiters and conv.getDelimiters accessor functions. In a future version of the Standard Library, the internal representation of this character set *will* change and any code that accesses it directly will break at that point. However, if you call conv.setDelimiters and conv.getDelimiters , you're guaranteed that the internal implementation will be hidden from you and your code will not fail when the internal representation changes.

`conv.getDelimiters( var Delims: cset );`

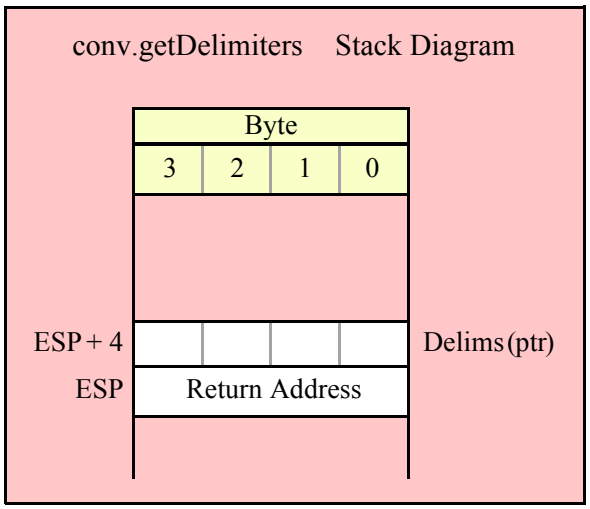
The conv.getDelimiters routine returns the current delimiters character set in the variable you pass by reference as the parameter. The *Delims* parameter is passed by reference (that is, you pass the address of the actual cset variable to receive the result). The following are examples of typical HLA high-level invocations of this routine looks like this:

```
conv.getDelimiters( saveDelims );

// EDI points at the delimiter cset:

conv.getDelimiters( [edi] );

// ptrtoDelims is a dword/pointer variable that contains
// the address of a cset object:

conv.getDelimiters( val ptrToDelims );
```

To call conv.getDelimiters using a low-level assembly syntax, you must push the address of a cset variable object onto the stack and then call the conv.getDelimiters function:

```
// saveDelims_s is a variable declared in the static/storage section:

pushd( &saveDelims_s );
call conv.getDelimiters;
```

```
// saveDelims_v is a variable declared in the var section or
// is a parameter:

lea( eax, saveDelims_v );
push( eax );
call conv.getDelimiters;

// Alternative call passing saveDelims_v if no 32-bit registers
// are available (this code assumes that EBP points at the current
// activation record/stack frame that contains saveDelims_v):

push( ebp );
add( @offset( saveDelims_v ), (type dword [esp] ));
call conv.getDelimiters;

// Low-level call assuming a 32-bit register (esi in this case)
// contains the address of the cset:

push( esi );
call conv.getDelimiters;

// Low-level call assuming a dword or pointer variable contains the
// address of the cset that will receive the delimiter character set:

push( ptrToDelims );
call conv.getDelimiters;
```



conv.setDelimiters(Delims: cset)

The conv.setDelimiters function lets you change the value of the internal delimiter character set. It requires a 16-byte character set parameter (passed by value) and will copy the value of this parameter to the internal character set variable. Note that this routine makes a copy of the actual parameter you pass it. If you pass an character set variable as the actual parameter, future changes to that character set variable (without a corresponding call to conv.setDelimiters) will have no effect on the internal delimiters character set that the standard library routines use. The following examples are typical HLL style calls to this function:

```
conv.setDelimiters( {' ', ','} );// Pass in a literal constant
conv.setDelimiters( csetVar );// Pass in a cset variable's value
conv.setDelimiters( [edx] );// EDX points at a cset variable
```

To call conv.setDelimiters using a low-level calling sequence, you'd first push the 16 bytes associated with the character set object (H.O. dword first down to the L.O. dword) and then call the conv.setDelimiters function. Here are some examples:

```
// Push the literal cset constant {' ', ','} onto the stack:

pushd( 0 ); // Must manually convert cset to a sequence of
pushd( $1001 );// four dwords (ugh!). Note: ORD( ' ' ) = $20
pushd( 0 ); // and ORD(',') = $2C so bit positions $20 and
pushd( 0 ); // $2c must contain '1's, zeros everywhere else.
call conv.setDelimiters;



// Push the cset variable "saveDelims" onto the stack and
// call conv.setDelimiters:

push( (type dword saveDelims[12]));
push( (type dword saveDelims[8]) );
push( (type dword saveDelims[4]) );
push( (type dword saveDelims[0]) );
call conv.setDelimiters.

// If manually converting a literal cset constant to the equivalent
// numeric values isn't your thing, you can also do the following
// (though this is slightly less efficient):

readonly
  spaceAndComma :cset := {' ', ',' };
endreadonly
push( (type dword spaceAndComma [12]));
push( (type dword spaceAndComma [8]) );
push( (type dword spaceAndComma [4]) );
push( (type dword spaceAndComma [0]) );
call conv.setDelimiters.
```

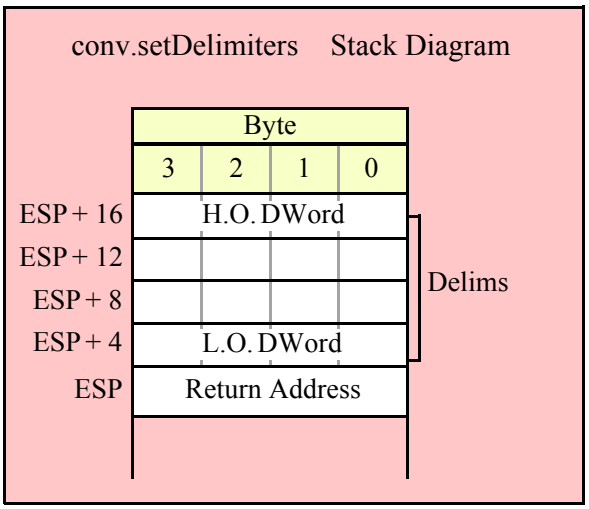
If you insist on using low-level calling sequences to call the conv.setDelimiters routine, you might want to consider writing a macro that will automatically push a literal cset constant for you. Here is a set of HLA macros that will do this task:

```
program t;

    // dword_n extracts the nth dword (0, 1, 2, 3) from a
    // 16-byte object such as a character set. cst must be
    // a cset constant value (or an lword), n must be an
    // integer constant in the range 0..3.

    #macro dword_n( cst, n );

        (
                (@byte( @lword(cst), n*4+3) << 24)
            |   (@byte( @lword(cst), n*4+2) << 16)
            |   (@byte( @lword(cst), n*4+1) << 8)
            |   (@byte( @lword(cst), n*4+0) << 0)
        )
    #endmacro

    // pushcset pushes the cset constant passed as an argument
    // onto the CPU's stack.  H.O. dword is pushed first, L.O.
    // dword is pushed last.
```

```
    #macro pushcset( cst );

        // Push the four dwords that make up a cset constant:

        pushd( dword_n( cst, 3 ) );
        pushd( dword_n( cst, 2 ) );
        pushd( dword_n( cst, 1 ) );
        pushd( dword_n( cst, 0 ) );

    #endmacro


begin t;

    // Example of pushcset invocation:

    pushcset( {' ', ','} );

end t;
```

Here is an example using conv.getDelimiters and conv.setDelimiters that demonstrates how to temporarily change the delimiters character set and the restore its value:

```
var
  saveDelims:cset;
      .
      .
      .
conv.getDelimiters( saveDelims );
conv.setDelimiters( { '!', '@' } )l
  .
  .
  .
conv.setDelimiters( saveDelims );
```



8.3 Hexadecimal Conversions

The standard library hexadecimal routines convert numeric values of varying sizes (8, 16, 32, 64, 80, and 128 bits) into a string of characters holding the hexadecimal representation of those values. The hexadecimal output routines can be broken down into the following categories:

Output type (string or sequence of characters to a buffer)

Fill type (no fill; fill with zeros to

a standardized length, based on data type; fill with a caller-specified character to a caller-specified length).

8.3.1 Internal Routines

The conversions module in the standard library contains several routines that are intended for internal use only. Generally, you should not call these routines directly from your application programs. These routines all have names that begin with an underscore. The internal hexadecimal conversion routines include: _hexTobuf64Size, _hexTobuf80, _hexTobuf80Size, _hexTobuf128, _hexTobuf128Size, _hexTobuf32, _hexTobuf32Size, and _hexTobuf64.

8.3.2 Hexadecimal Numeric Size Functions

The hexadecimal conversion size functions return the number of digit print positions required by the conversion of a numeric value to a hexadecimal string. There are two sets of six routines that compute the output size: one set computes the fixed-size width and the other set computes a varying-sized width.

8.3.2.1 Fixed Size Hexadecimal Size Functions

It is common practice to display hexadecimal values using exactly one digit for each nibble of the corresponding data type, including leading zeros, as necessary. The common fixed sizes are byte=2, word=4, dword=8, qword=16, tbyte=20, and lword=32. With underscore output enabled (see conv.setUnderscores) these values are byte=2, word=4, dword=9, qword=19, tb=24, and lword=39. Because these numbers are fixed (at least, for a given underscores flag setting) there are only three reasons for calling these functions:

- You don't know the underscores flag setting when executing a particular section of code (which can affect the output size of dword, qword, tbyte, and lword objects), or,
- You're generating a call to these functions via some macro that is given a function name like "putb" or "puth8" and you're manually constructing the size function to call via the assembler's compile-time language. or,
- You're writing generic code and you want to make it easy to modify the code in the future.

```
procedure conv.bSize( b:byte in al ); @returns( "eax" );
```

This function always returns 2 because the fixed output size of a byte is two hexadecimal digits (two 4-bit nibbles).

```
HLA high-level calling sequence examples:

conv.bSize( byteVariable );
conv.bSize( <byte register> ); // al, ah, bl, bh, cl, ch, dl, dh
conv.bSize( <constant> );      // Must fit into eight bits
```

Because conv.bSize passes its input parameter in the AL register, any form of the high-level calling sequence except "conv.bSize(al);" will automatically generate an instruction of the form "mov*(<operand>*,al);". Therefore, if at all possible, you should try to have the value whose size you wish to compute already sitting in the AL register and pass AL as the parameter to conv.bSize.

```
HLA low-level calling sequence examples:

mov( byteVariable, al );
call conv.bSize;

mov( <byte register>, al ); // ah, bl, bh, cl, ch, dl, dh
call conv.bSize;

mov( <constant>, al );
call conv.bSize;
```

procedure conv.wSize(w:word in ax); @returns("eax");

This function always returns 4 because the "natural" size of a word is four hexadecimal digits (four 4-bit nibbles).

```
HLA high-level calling sequence examples:

conv.wSize( wordVariable );
conv.wSize( <word register> ); // ax, bx, cx, dx, bp, sp, si, di
conv.wSize( <constant> );      // Must fit into 16 bits
```

Because conv.wSize passes its input parameter in the AX register, any form of the high-level calling sequence except "conv.wSize(ax);" will automatically generate an instruction of the form "mov*(<operand>*,ax);". Therefore, if at all possible, you should try to have the value whose size you wish to compute already sitting in the AX register and pass AX as the parameter to conv.wSize.

```
HLA low-level calling sequence examples:

mov( wordVariable, ax );
call conv.wSize;

mov( <word register>, ax ); // bx, cx, dx, bp, sp, si, di
call conv.wSize;

mov( <constant>, ax );
call conv.wSize;
```

procedure conv.dSize(d:dword in eax); @returns("eax");

This function returns 8 if the internal underscores flag is false, 9 if it is true, because the "natural" size of a double word is eight hexadecimal digits (eight 4-bit nibbles).

```
HLA high-level calling sequence examples:

conv.dSize( dwordVariable );
conv.dSize( <dword register> ); // eax, ebx, ecx, edx,
                                //ebp, esp, esi, edi
conv.dSize( <constant> );       // Must fit into 32 bits
```

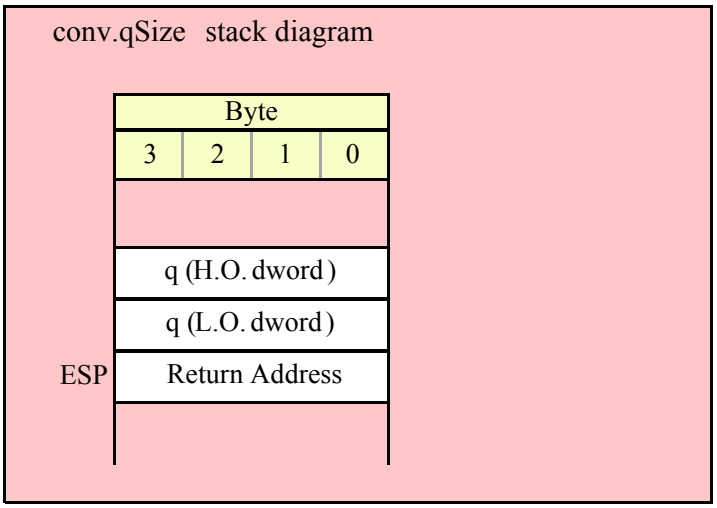
Because conv.dSize passes its input parameter in the EAX register, any form of the high-level calling sequence except "conv.dSize(eax);" will automatically generate an instruction of the form "mov*(<operand>*,eax);". Therefore, if at all possible, you should try to have the value whose size you wish to compute already sitting in the EAX register and pass EAX as the parameter to conv.dSize.

```
HLA low-level calling sequence examples:

mov( dwordVariable, eax );
call conv.dSize;

mov( <dword register>, eax ); // ebx, ecx, edx, ebp, esp, esi, edi
call conv.dSize;

mov( <constant>, eax );
call conv.dSize;
```

procedure conv.qSize(q:qword); @returns("eax");

This function returns 16 if the internal underscores flag is false, 19 if it is true, because the "natural" size of a quad word is 16 hexadecimal digits (16 4-bit nibbles) and there are four groups of four digits with underscores between them (if the underscores flag contains true).

```
HLA high-level calling sequence examples:

conv.qSize( qwordVariable );
conv.qSize( <constant> );       // Must fit into 64 bits


HLA low-level calling sequence examples:

// Passing a qword variable

push( (type dword qwordVar[4]) );// Push H.O. dword first
push( (type dword qwordVar[0]) );   // Push L.O. dword second
call conv.qSize;

// Example where 64-bit value is held in EDX:EAX

push( edx ); // Push H.O. dword first
push( eax ); // Push L.O. dword second
call conv.qSize;

// Passing a qword constant:

pushd( <qword constant> >> 32 );          // Push H.O. dword
pushd( <qword constant> & $FFFF_FFFF );  // Push L.O. dword
call conv.qSize;
```

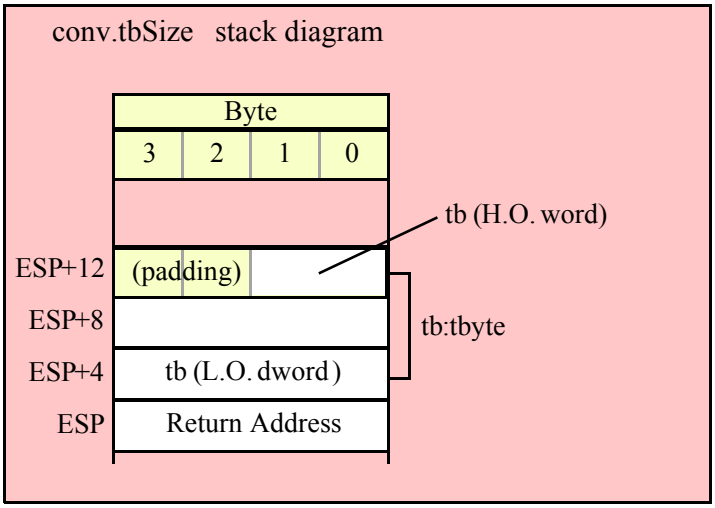


procedure conv.tbSize(tb:tbyte); @returns("eax");

This function returns 20 if the internal underscores flag is false, 24 if it is true, because the "natural" size of a ten-byte word is 20 hexadecimal digits (20 4-bit nibbles) and there are five groups of four digits with underscores between them (if the underscores flag contains true).

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

```
conv.tbSize( tbyteVariable );
conv.tbSize( <constant> );       // Must fit into 80 bits


HLA low-level calling sequence examples:

// Passing a tbyte variable

pushw( 0 );                          // Must pad parameter to 12 bytes
push( (type word tbyteVar[8]) );     // Push H.O. word
push( (type dword tbyteVar[4]) );// Push middle dword second
push( (type dword tbyteVar[0]) );    // Push L.O. dword third
call conv.tbSize;

// Passing a tbyte constant:

pushw( 0 ); // Must pad to 12 bytes.
pushw( <tbyte constant> >> 64 );                // Push H.O. word
pushd( (<tbyte constant> >> 32) & $FFFF_FFFF ); // Push middle dword
pushd( <tbyte constant> & $FFFF_FFFF );         // Push L.O. dword
call conv.tbSize;
```

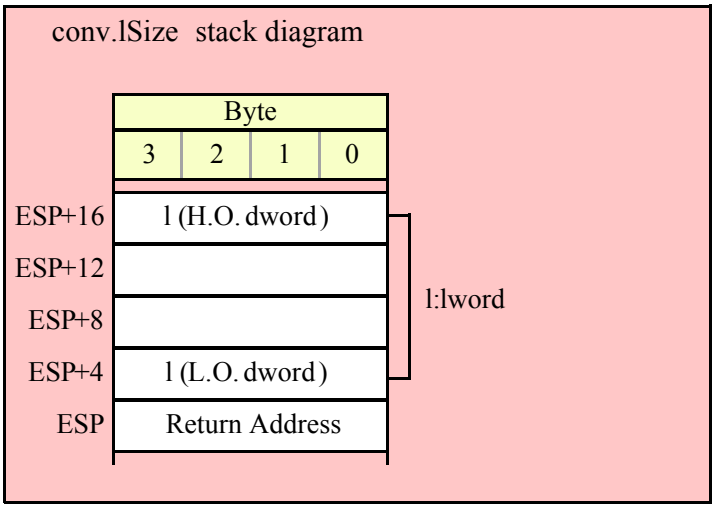


procedure conv.lSize(l:lword); @returns("eax");

This function returns 32 if the internal underscores flag is false, 39 if it is true, because the "natural" size of an lword is 32 hexadecimal digits (32 4-bit nibbles) and there are eight groups of four digits with underscores between them (if the underscores flag contains true).

```
HLA high-level calling sequence examples:

conv.lSize( lwordVariable );
conv.lSize( <constant> );       // Must fit into 128 bits


HLA low-level calling sequence examples:

// Passing an lword variable

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

```
push( (type dword tbyteVar[8]) );
push( (type dword tbyteVar[4]) );
push( (type dword tbyteVar[0]) );   // Push L.O. dword last
call conv.lSize;

// Passing a lword constant:

pushd( <lword constant> >> 96  );  // Push H.O. dword first
pushw( (<lword constant> >> 64) & $FFFF_FFFF );
pushd( (<lword constant> >> 32) & $FFFF_FFFF );
pushd( <lword constant> & $FFFF_FFFF );  // Push L.O. dword last
call conv.lSize;
```



8.3.2.2 Standard Hexadecimal Size Functions

The h8Size, h16Size, h32Size, h64Size, h80Size, and h128Size routines compute the minimum number of output hexadecimal digits (with no leading zeros). These functions return a count that includes space for underscores if the internal underscores flag contains true (see conv.setUnderscores for details).

```
procedure conv.h8Size( b:byte in al ); @returns( "eax" );
```

This function returns the number of print positions required by the conversion of the value in AL to a string of hexadecimal digits. The return value is always 1 or 2 (as a single byte never consumes more than two hexadecimal digits). Note that the internal underscores flag setting does not affect the return result because the conversion routines never inject an underscore into a hexadecimal conversion unless there are at least five output digits.

```
HLA high-level calling sequence examples:

conv.h8Size( byteVariable );
conv.h8Size( <byte register> ); // al, ah, bl, bh, cl, ch, dl, dh
conv.h8Size( <constant> );      // Must fit into eight bits
```

Because conv.h8Size passes its input parameter in the AL register, any form of the high-level calling sequence except "conv.h8Size(al);" will automatically generate an instruction of the form "mov*(<operand>*,al);". Therefore, if at all possible, you should try to have the value whose size you wish to compute already sitting in the AL register and pass AL as the parameter to conv.h8Size.

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

```
mov( byteVariable, al );
call conv.h8Size;

mov( <byte register>, al ); // ah, bl, bh, cl, ch, dl, dh
call conv.h8Size;

mov( <constant>, al );
call conv.h8Size;
```

procedure conv.h16Size(w:word in ax); @returns("eax");

This function returns the number of print positions required by the conversion of the value in AX to a string of hexadecimal digits. The return value is always 1, 2, 3, or 4 (as a single word never requires more than four hexadecimal digits). Note that the internal underscores flag setting does not affect the return result because the conversion routines never inject an underscore into a hexadecimal conversion unless there are at least five output digits.

```
HLA high-level calling sequence examples:

conv.h16Size( wordVariable );
conv.h16Size( <word register> ); // ax, bx, cx, dx, bp, sp, si, di
conv.h16Size( <constant> );      // Must fit into 16 bits
```

Because conv.h16Size passes its input parameter in the AX register, any form of the high-level calling sequence except "conv.h16Size(ax);" will automatically generate an instruction of the form "mov*(<operand>*,ax);". Therefore, if at all possible, you should try to have the value whose size you wish to compute already sitting in the AX register and pass AX as the parameter to conv.h16Size.

```
HLA low-level calling sequence examples:

mov( wordVariable, ax );
call conv.h16Size;

mov( <word register>, ax ); // bx, cx, dx, bp, sp, si, di
call conv.h16Size;

mov( <constant>, ax );
call conv.h16Size;
```

procedure conv.h32Size(s:dword in eax); @returns("eax");

This function returns the number of print positions required by the conversion of the value in EAX to a string of hexadecimal digits. The return value is always in the range 1-8 if the internal underscores flag is false, it is in the range 1-9 if the internal underscores flag is true (see the discussion of conv.setUnderscores for details).

```
HLA high-level calling sequence examples:

conv.h32Size( dwordVariable );
conv.h32Size( <dword register> ); // eax, ebx, ecx, edx,
                                  //ebp, esp, esi, edi
conv.h32Size( <constant> );       // Must fit into 32 bits
```

Because conv.h32Size passes its input parameter in the EAX register, any form of the high-level calling sequence except "conv.h32Size(eax);" will automatically generate an instruction of the form "mov*(<operand>*,eax);". Therefore, if at all possible, you should try to have the value whose size you wish to compute already sitting in the EAX register and pass EAX as the parameter to conv.h32Size.

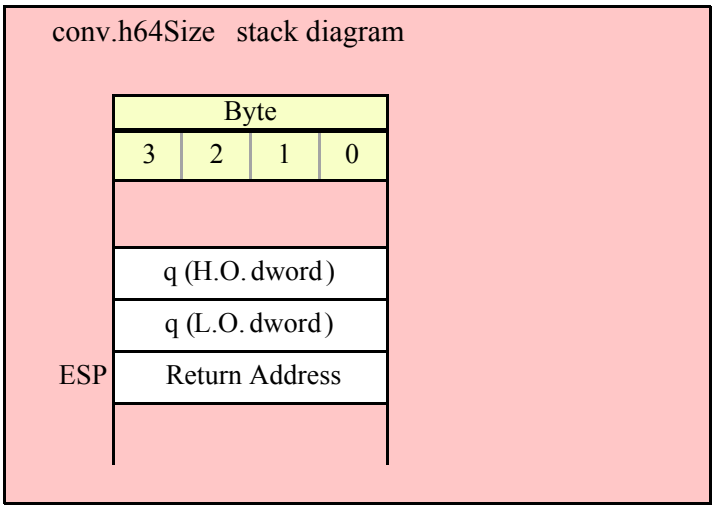
```
HLA low-level calling sequence examples:
```

```
mov( dwordVariable, eax );
call conv.h32Size;

mov( <dword register>, eax ); // ebx, ecx, edx, ebp, esp, esi, edi
call conv.h32Size;

mov( <constant>, eax );
call conv.h32Size;
```

procedure conv.h64Size(q:qword); @returns("eax");

This function returns the number of print positions required by the conversion of the value passed in q to a string of hexadecimal digits. The return value is always in the range 1-16 if the internal underscores flag is false, it is in the range 1-19 if the internal underscores flag is true (see the discussion of conv.setUnderscores for details).

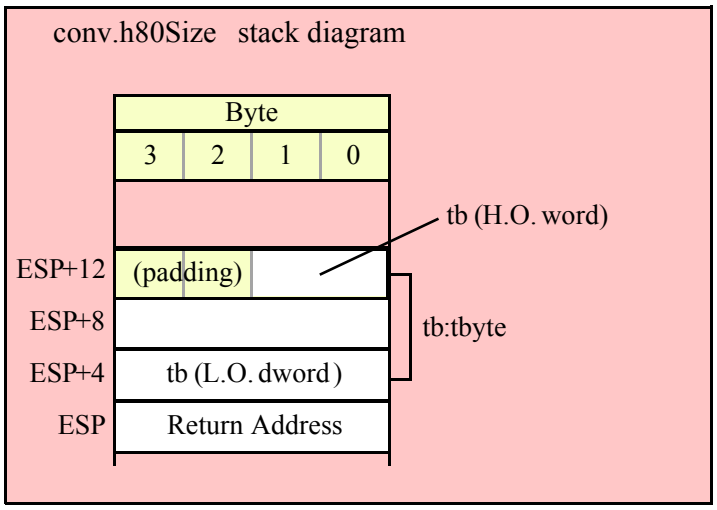
```
HLA high-level calling sequence examples:

conv.h64Size( qwordVariable );
conv.h64Size( <constant> );       // Must fit into 64 bits


HLA low-level calling sequence examples:

// Passing a qword variable

push( (type dword qwordVar[4]) );// Push H.O. dword first
push( (type dword qwordVar[0]) );   // Push L.O. dword second
call conv.h64Size;

// Example where 64-bit value is held in EDX:EAX

push( edx ); // Push H.O. dword first
push( eax ); // Push L.O. dword second
call conv.h64Size;

// Passing a qword constant:

pushd( <qword constant> >> 32 );         // Push H.O. dword
pushd( <qword constant> & $FFFF_FFFF );  // Push L.O. dword
call conv.h64Size;
```



procedure conv.h80Size(tb:tbyte); @returns("eax");

This function returns the number of print positions required by the conversion of the value passed in tb to a string of hexadecimal digits. The return value is always in the range 1-20 if the internal underscores flag is false, it is in the range 1-24 if the internal underscores flag is true (see the discussion of conv.setUnderscores for details).

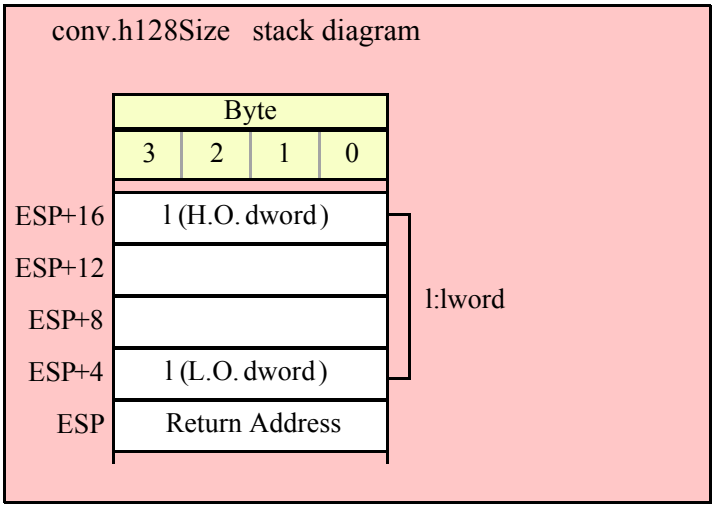
```
HLA high-level calling sequence examples:

conv.h80Size( tbyteVariable );
conv.h80Size( <constant> );       // Must fit into 80 bits


HLA low-level calling sequence examples:

// Passing a tbyte variable

pushw( 0 );                          // Must pad parameter to 12 bytes
push( (type word tbyteVar[8]) );     // Push H.O. word
push( (type dword tbyteVar[4]) );// Push middle dword second
push( (type dword tbyteVar[0]) );    // Push L.O. dword third
call conv.h80Size;

// Passing a tbyte constant:

pushw( 0 ); // Must pad to 12 bytes.
pushw( <tbyte constant> >> 64 );                // Push H.O. word
pushd( (<tbyte constant> >> 32) & $FFFF_FFFF ); // Push middle dword
pushd( <tbyte constant> & $FFFF_FFFF );         // Push L.O. dword
call conv.h80Size;
```



procedure conv.h128Size(l:lword); @returns("eax");

This function returns the number of print positions required by the conversion of the value passed in l to a string of hexadecimal digits. The return value is always in the range 1-32 if the internal underscores flag is false, it is in the range 1-39 if the internal underscores flag is true (see the discussion of conv.setUnderscores for details).

```
HLA high-level calling sequence examples:

conv.h128Size( lwordVariable );
conv.h128Size( <constant> );       // Must fit into 128 bits


HLA low-level calling sequence examples:

// Passing an lword variable

push( (type dword lwordVar[12]));   // Push H.O. dword first
push( (type dword tbyteVar[8]) );
push( (type dword tbyteVar[4]) );
push( (type dword tbyteVar[0]) );   // Push L.O. dword last
call conv.h128Size;

// Passing a lword constant:

pushd( <lword constant> >> 96  );  // Push H.O. dword first
pushw( (<lword constant> >> 64) & $FFFF_FFFF );
pushd( (<lword constant> >> 32) & $FFFF_FFFF );
pushd( <lword constant> & $FFFF_FFFF );  // Push L.O. dword last
call conv.h128Size;
```



8.3.3 Hexadecimal Numeric to Buffer Conversions

The hexadecimal numeric to buffer conversion routines translate a numeric value to a sequence of hexadecimal characters and store those characters into memory starting at the address pointed at by the EDI register. After the conversion, the EDI register points at the first byte in memory beyond the sequence of characters these routines produce. With successive calls to these routines (or any routine that emits characters to the buffer at which EDI points) you can build up larger strings.

If the internal underscores flag is true, these routines will emit an underscore between each group of four hexadecimal digits.

8.3.3.1 Fixed Length Hexadecimal Numeric to Buffer Conversions

The fixed length hexadecimal to buffer conversion routines translate an input numeric value to a fixed-length string (depending on the data type size and the setting of the internal underscores flag). These functions emit the characters of the string (including leading zeros, as necessary) to sequential locations starting at the address held in EDI.

```
procedure conv.bToBuf( b:byte in al; var buffer:var in edi );
```

Converts the numeric value in AL to a sequence of exactly two hexadecimal digits (including a leading zero if the value is in the range $0-$f) and stores these two characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. Because the conversion is always less than five digits, the internal underscores flag does not affect the output this function produces.

```
HLA high-level calling sequence examples:

// The following will load "byteVariable" into AL and
// the address of "charArrayVariable" into EDI and then
// call conv.bToBuf:

conv.bToBuf( byteVariable, charArrayVariable );

// The following call will copy BH into AL and
// EDX into EDI prior to calling conv.bToBuf:

conv.bToBuf( bh, [edx] );

// The following just calls conv.bToBuf as AL and EDI
// already hold the parameter values:
```

```
conv.bToBuf( al, [edi] );

// The following loads the constant in AL and calls
// conv.bToBuf:

conv.bToBuf( <constant>, [edi] ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the AL and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. byte of EAX (i.e., AL) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AL and [EDI].

```
HLA low-level calling sequence examples:

// Passing a byte variable and a buffer variable:

mov( byteVariable, al );
lea( edi, charArrayVariable );
call conv.bToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( byteVariable, al );
mov( &charArrayVariable, edi );
call conv.bToBuf;

// Passing a pair of registers (that are not
// AL and EDI):

mov( bh, al );
mov( edx, edi );
call conv.bToBuf;

// Passing a constant:

mov( <constant>, al );
call conv.bToBuf; // Assume EDI already contains buffer address.
```

procedure conv.wToBuf(w:word in ax; var buffer:var in edi);

Converts the numeric value in AX to a sequence of exactly four hexadecimal digits (including leading zeros, as necessary) and stores these four characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. Because the conversion is always less than five digits, the internal underscores flag does not affect the output this function produces.

```
HLA high-level calling sequence examples:

// The following will load "wordVariable" into AX and
// the address of "charArrayVariable" into EDI and then
// call conv.wToBuf:

conv.wToBuf( wordVariable, charArrayVariable );

// The following call will copy BX into AX and
// EDX into EDI prior to calling conv.wToBuf:

conv.wToBuf( bx, [edx] );
```

```
// The following just calls conv.wToBuf as AX and EDI
// already hold the parameter values:

conv.wToBuf( ax, [edi] );

// The following loads the constant in AX and calls
// conv.wToBuf:

conv.wToBuf( <constant>, [edi] ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the AX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. word of EAX (i.e., AX) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AX and [EDI].

```
HLA low-level calling sequence examples:

// Passing a word variable and a buffer variable:

mov( wordVariable, ax );
lea( edi, charArrayVariable );
call conv.wToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( wordVariable, ax );
mov( &charArrayVariable, edi );
call conv.wToBuf;

// Passing a pair of registers (that are not
// AX and EDI):

mov( bx, ax );
mov( edx, edi );
call conv.wToBuf;

// Passing a constant:

mov( <constant>, ax );
call conv.wToBuf; // Assume EDI already contains buffer address.
```

procedure conv.dToBuf(d:dword in eax; var buffer:var in edi);

Converts the numeric value in EAX to a sequence of exactly eight hexadecimal digits (including leading zeros, as necessary) and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. If the internal underscores flag contains true, this function will emit a nine-character string with a single underscore between the fourth and fifth digits.

```
HLA high-level calling sequence examples:

// The following will load "dwordVariable" into EAX and
// the address of "charArrayVariable" into EDI and then
// call conv.dToBuf:

conv.dToBuf( dwordVariable, charArrayVariable );

// The following call will copy EBX into EAX and
```

```
// EDX into EDI prior to calling conv.dToBuf:

conv.dToBuf( ebx, [edx] );

// The following just calls conv.dToBuf as EAX and EDI
// already hold the parameter values:

conv.dToBuf( eax, [edi] );

// The following loads the constant in EAX and calls
// conv.dToBuf:

conv.dToBuf( <constant>, [edi] ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EAX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite EAX before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not EAX and [EDI].

```
HLA low-level calling sequence examples:

// Passing a dword variable and a buffer variable:

mov( dwordVariable, eax );
lea( edi, charArrayVariable );
call conv.dToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( dwordVariable, eax );
mov( &charArrayVariable, edi );
call conv.dToBuf;

// Passing a pair of registers (that are not
// EAX and EDI):

mov( ebx, eax );
mov( edx, edi );
call conv.dToBuf;

// Passing a constant:

mov( <constant>, eax );
call conv.dToBuf; // Assume EDI already contains buffer address.
```

procedure conv.qToBuf(q:qword; var buffer:var in edi);

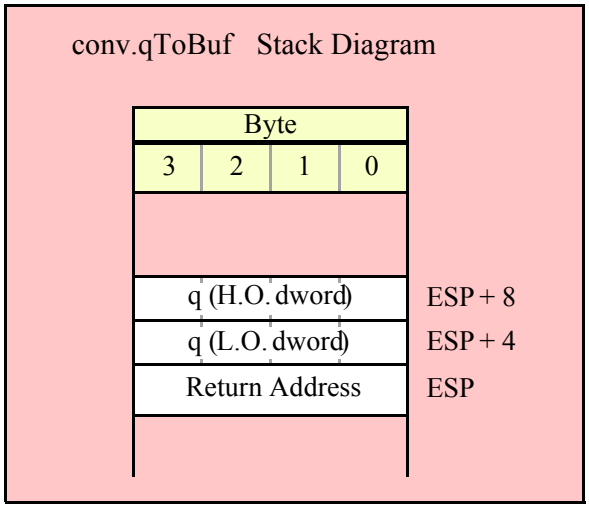
Converts the numeric value passed in q to a sequence of exactly 16 hexadecimal digits (including leading zeros, as necessary) and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. If the internal underscores flag contains true, this function will emit a 19-character string with underscores between the 4^{th} and 5^{th}, 8^{th} and 9^{th}, and 12^{th} and 13^{th} digits.

```
HLA high-level calling sequence examples:

// The following will push the value of "qwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.qToBuf:
```

```
conv.qToBuf( qwordVariable, charArrayVariable );


// The following pushes the constant onto the stack and calls
// conv.qToBuf:

conv.qToBuf( <constant>, [edi] ); // <constant> must fit in 64 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

```
HLA low-level calling sequence examples:

// Passing a qword variable and a buffer variable:

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
lea( edi, charArrayVariable );
call conv.qToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
mov( &charArrayVariable, edi );
call conv.qToBuf;

// Passing a constant:

pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
call conv.qToBuf; // Assume EDI already contains buffer address.
```



procedure conv.tbToBuf(tb:tbyte; var buffer:var in edi);

Converts the numeric value passed in tb to a sequence of exactly 20 hexadecimal digits (including leading zeros, as necessary) and stores these characters at the location pointed at by EDI. This function returns EDI

pointing at the first byte after the sequence. If the internal underscores flag contains true, this function will emit a 24-character string with underscores between the 4th and 5th, 8th and 9th, 12th and 13th , and 16th and 17th digits.

```
HLA high-level calling sequence examples:

// The following will push the value of "tbyteVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.tbToBuf:

conv.tbToBuf( tbyteVariable, charArrayVariable );
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

```
HLA low-level calling sequence examples:

// Passing a tbyte variable and a buffer variable:

pushw( 0 ); // Must pad parameter to 12 bytes
push( (type word tbyteVariable[8]));   // H.O. word first
  push( (type dword tbyteVariable[4]));
  push( (type dword qwordVariable[0]));// L.O. dword last
lea( edi, charArrayVariable );
call conv.tbToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

pushw( 0 ); // Must pad parameter to 12 bytes
push( (type word tbyteVariable[8]));   // H.O. word first
  push( (type dword tbyteVariable[4]));
  push( (type dword qwordVariable[0]));// L.O. dword last
mov( &charArrayVariable, edi );
call conv.tbToBuf;

// Passing a constant:

pushd( <constant> >> 64 ); // Push H.O. word as dword, first
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
call conv.tbToBuf; // Assume EDI already contains buffer address.
```

procedure conv.lToBuf(l:lword; var buffer:var in edi);

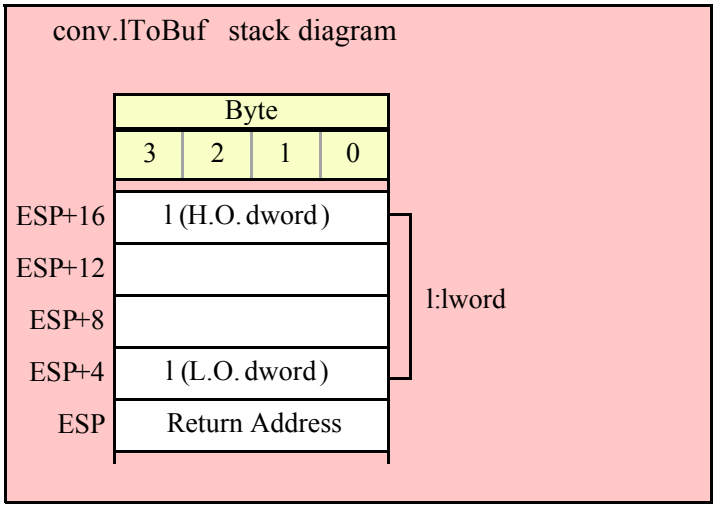
Converts the numeric value passed in l to a sequence of exactly 32 hexadecimal digits (including leading zeros, as necessary) and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. If the internal underscores flag contains true, this function will emit a 39 character string with underscores between the 4th and 5th, 8th and 9th, 12th and 13th , 16th and 17th, 20th and 21st, 24th and 25th, and 28th and 29th digits.

```
HLA high-level calling sequence examples:

// The following will push the value of "lwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.lToBuf:
```

```
conv.lToBuf( lwordVariable, charArrayVariable );


// The following pushes the constant onto the stack and calls
// conv.lToBuf:

conv.lToBuf( <constant>, [edi] ); // <constant> must fit in 128 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

```
HLA low-level calling sequence examples:

// Passing an lword variable and a buffer variable:

  push( (type dword lwordVariable[12]));// H.O. dword first
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
lea( edi, charArrayVariable );
call conv.lToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

  push( (type dword lwordVariable[12]));// H.O. dword first
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
mov( &charArrayVariable, edi );
call conv.lToBuf;

// Passing a constant:

pushd( <constant> >> 96 );// Push H.O. dword of constant first.
pushd( (<constant> >> 64)& $FFFF_FFFF );
pushd( (<constant> >> 32)& $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
call conv.lToBuf; // Assume EDI already contains buffer address.
```



8.3.3.2 Variable Length Hexadecimal Numeric to Buffer Conversions

The variable length hexadecimal to buffer conversion routines translate an input numeric value to a variable-length string (depending on the value, data type size, and the setting of the internal underscores flag). These functions emit the characters of the string (without leading zeros) to sequential locations starting at the address held in EDI.

procedure conv.h8ToBuf(b:byte in al; var buffer:var in edi);

Converts the numeric value in AL to a sequence of one or two hexadecimal digits and stores these two characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. Because the conversion is always less than five digits, the internal underscores flag does not affect the output this function produces.

```
HLA high-level calling sequence examples:

// The following will load "byteVariable" into AL and
// the address of "charArrayVariable" into EDI and then
// call conv.h8ToBuf:

conv.h8ToBuf( byteVariable, charArrayVariable );

// The following call will copy BH into AL and
// EDX into EDI prior to calling conv.h8ToBuf:

conv.h8ToBuf( bh, [edx] );

// The following just calls conv.h8ToBuf as AL and EDI
// already hold the parameter values:

conv.h8ToBuf( al, [edi] );

// The following loads the constant in AL and calls
// conv.h8ToBuf:

conv.h8ToBuf( <constant>, [edi] ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the AL and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. byte of EAX (i.e., AL) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AL and [EDI].

```
HLA low-level calling sequence examples:

// Passing a byte variable and a buffer variable:

mov( byteVariable, al );
lea( edi, charArrayVariable );
call conv.h8ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( byteVariable, al );
mov( &charArrayVariable, edi );
call conv.h8ToBuf;

// Passing a pair of registers (that are not
// AL and EDI):
```

```
mov( bh, al );
mov( edx, edi );
call conv.h8ToBuf;

// Passing a constant:

mov( <constant>, al );
call conv.h8ToBuf; // Assume EDI already contains buffer address.
```

procedure conv.h16ToBuf(w:word in ax; var buffer:var in edi);

Converts the numeric value in AX to a sequence of one to four hexadecimal digits and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. Because the conversion is always less than five digits, the internal underscores flag does not affect the output this function produces.

```
HLA high-level calling sequence examples:

// The following will load "wordVariable" into AX and
// the address of "charArrayVariable" into EDI and then
// call conv.h16ToBuf:

conv.h16ToBuf( wordVariable, charArrayVariable );

// The following call will copy BX into AX and
// EDX into EDI prior to calling conv.h16ToBuf:

conv.h16ToBuf( bx, [edx] );

// The following just calls conv.h16ToBuf as AX and EDI
// already hold the parameter values:

conv.h16ToBuf( ax, [edi] );

// The following loads the constant in AX and calls
// conv.h16ToBuf:

conv.h16ToBuf( <constant>, [edi] ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the AX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. word of EAX (i.e., AX) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AX and [EDI].

```
HLA low-level calling sequence examples:

// Passing a word variable and a buffer variable:

mov( wordVariable, ax );
lea( edi, charArrayVariable );
call conv.h16ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( wordVariable, ax );
mov( &charArrayVariable, edi );
call conv.h16ToBuf;
```

```
// Passing a pair of registers (that are not
// AX and EDI):

mov( bx, ax );
mov( edx, edi );
call conv.h16ToBuf;

// Passing a constant:

mov( <constant>, ax );
call conv.h16ToBuf; // Assume EDI already contains buffer address.
```

procedure conv.h32ToBuf(d:dword in eax; var buffer:var in edi);

Converts the numeric value in EAX to a sequence of one to eight hexadecimal digits and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. If the internal underscores flag contains true and the value is $1_0000 or greater, then this function will emit an underscore between the fourth and fifth digits in the output string.

```
HLA high-level calling sequence examples:

// The following will load "dwordVariable" into EAX and
// the address of "charArrayVariable" into EDI and then
// call conv.h32ToBuf:

conv.h32ToBuf( dwordVariable, charArrayVariable );

// The following call will copy EBX into EAX and
// EDX into EDI prior to calling conv.h32ToBuf:

conv.h32ToBuf( ebx, [edx] );

// The following just calls conv.h32ToBuf as EAX and EDI
// already hold the parameter values:

conv.h32ToBuf( eax, [edi] );

// The following loads the constant in EAX and calls
// conv.h32ToBuf:

conv.h32ToBuf( <constant>, [edi] ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EAX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite EAX before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not EAX and [EDI].

```
HLA low-level calling sequence examples:

// Passing a dword variable and a buffer variable:

mov( dwordVariable, eax );
lea( edi, charArrayVariable );
call conv.h32ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):
```

```
mov( dwordVariable, eax );
mov( &charArrayVariable, edi );
call conv.h32ToBuf;

// Passing a pair of registers (that are not
// EAX and EDI):

mov( ebx, eax );
mov( edx, edi );
call conv.h32ToBuf;

// Passing a constant:

mov( <constant>, eax );
call conv.h32ToBuf; // Assume EDI already contains buffer address.
```

procedure conv.h64ToBuf(q:qword; var buffer:var in edi);

Converts the numeric value in q to a sequence of 1 to 16 hexadecimal digits and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence.

If the internal underscores flag contains true and the value is $1_0000 or greater, then this function will emit an underscore between the 4th and 5th digits in the output string.

If the value is $1_0000 or greater, then this function will emit an underscore between the 4th and 5th digits in the output string.

If the value is $1_0000_0000 or greater, then this function will emit an underscore between the 8th and 9th digits.

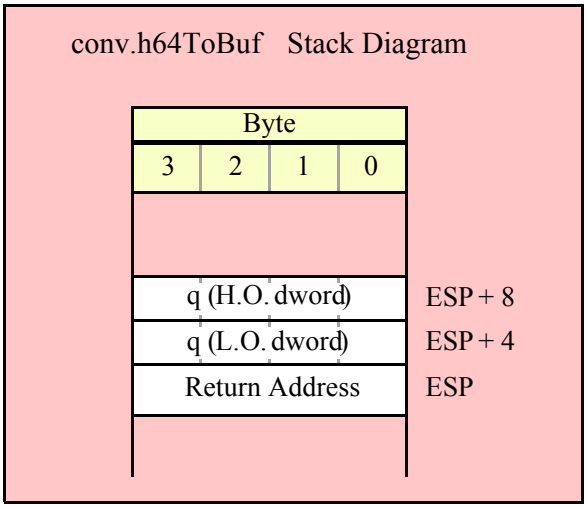
If the value is $1_0000_0000_0000 or greater, then this function will emit an underscore between the 12th and 13th digits.

```
HLA high-level calling sequence examples:

// The following will push the value of "qwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.h64ToBuf:

conv.h64ToBuf( qwordVariable, charArrayVariable );


// The following pushes the constant onto the stack and calls
// conv.h64ToBuf:

conv.h64ToBuf( <constant>, [edi] ); // <constant> must fit in 64 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

```
HLA low-level calling sequence examples:

// Passing a qword variable and a buffer variable:

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
lea( edi, charArrayVariable );
call conv.h64ToBuf;

// Alternate form of above if charArrayVariable is
```

```
// a static object (STATIC, READONLY, or STORAGE):

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
mov( &charArrayVariable, edi );
call conv.h64ToBuf;

// Passing a constant:

pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
call conv.h64ToBuf; // Assume EDI already contains buffer address.
```



procedure conv.h80ToBuf(tb:tbyte; var buffer:var in edi);

Converts the numeric value in tb to a sequence of 1 to 20 hexadecimal digits and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence.

If the internal underscores flag contains true, and:

If the value is $1_0000 or greater, then this function will emit an underscore between the 4th and 5th digits in the output string.

If the value is $1_0000_0000 or greater, then this function will emit an underscore between the 8th and 9th digits.

If the value is $1_0000_0000_0000 or greater, then this function will emit an underscore between the 12th and 13th digits.

If the value is $1_0000_0000_0000_0000 or greater, then this function will emit an underscore between the 16th and 17th digits.

```
HLA high-level calling sequence examples:

// The following will push the value of "tbyteVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.h80ToBuf:

conv.h80ToBuf( tbyteVariable, charArrayVariable );
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

```
HLA low-level calling sequence examples:

// Passing a tbyte variable and a buffer variable:

pushw( 0 ); // Must pad parameter to 12 bytes
push( (type word tbyteVariable[8]));   // H.O. word first
  push( (type dword tbyteVariable[4]));
  push( (type dword qwordVariable[0]));// L.O. dword last
lea( edi, charArrayVariable );
call conv.h80ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

pushw( 0 ); // Must pad parameter to 12 bytes
push( (type word tbyteVariable[8]));   // H.O. word first
  push( (type dword tbyteVariable[4]));
  push( (type dword qwordVariable[0]));// L.O. dword last
mov( &charArrayVariable, edi );
call conv.h80ToBuf;

// Passing a constant:

pushd( <constant> >> 64 ); // Push H.O. word as dword, first
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
call conv.tbToBuf; // Assume EDI already contains buffer address.
```

procedure conv.h128ToBuf(l:lword; var buffer:var in edi);

Converts the numeric value in l to a sequence of 1 to 32 hexadecimal digits and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence.

If the internal underscores flag contains true, and:

If the value is $1_0000 or greater, then this function will emit an underscore between the 4th and 5th digits in the output string.

If the value is $1_0000_0000 or greater, then this function will emit an underscore between the 8th and 9th digits.

If the value is $1_0000_0000_0000 or greater, then this function will emit an underscore between the 12th and 13th digits.

If the value is $1_0000_0000_0000_0000 or greater, then this function will emit an underscore between the 16th and 17th digits.

If the value is $1_0000_0000_0000_0000_0000 or greater, then this function will emit an underscore between the 20th and 21st digits.

If the value is $1_0000_0000_0000_0000_0000_0000 or greater, then this function will emit an underscore between the 24th and 25th digits.

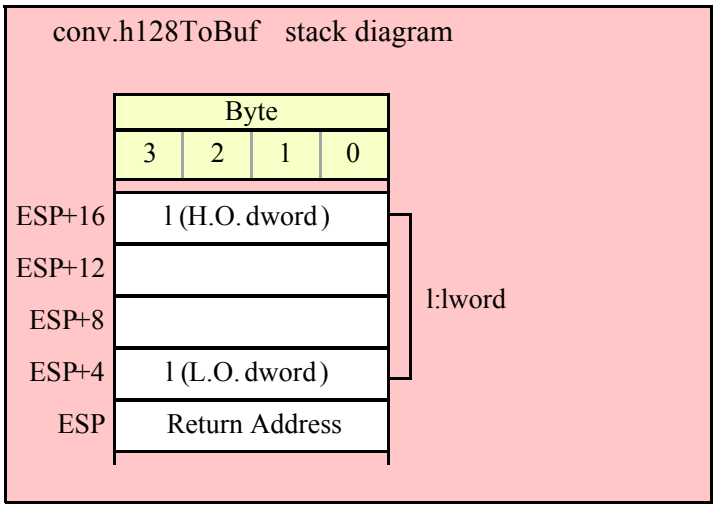
If the value is $1_0000_0000_0000_0000_0000_0000_0000 or greater, then this function will emit an underscore between the 28th and 29th digits.

```
HLA high-level calling sequence examples:

// The following will push the value of "lwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.h128ToBuf:

conv.h128ToBuf( lwordVariable, charArrayVariable );
```

```
// The following pushes the constant onto the stack and calls
// conv.h128ToBuf:

conv.h128ToBuf( <constant>, [edi] ); // <constant> must fit in 128 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

```
HLA low-level calling sequence examples:

// Passing an lword variable and a buffer variable:

  push( (type dword lwordVariable[12]));// H.O. dword first
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
lea( edi, charArrayVariable );
call conv.h128ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

  push( (type dword lwordVariable[12]));// H.O. dword first
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
mov( &charArrayVariable, edi );
call conv.h128ToBuf;

// Passing a constant:

pushd( <constant> >> 96 );// Push H.O. dword of constant first.
pushd( (<constant> >> 64)& $FFFF_FFFF );
pushd( (<constant> >> 32)& $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
call conv.h128ToBuf; // Assume EDI already contains buffer address.
```



8.3.4 Hexadecimal Numeric to String Conversions

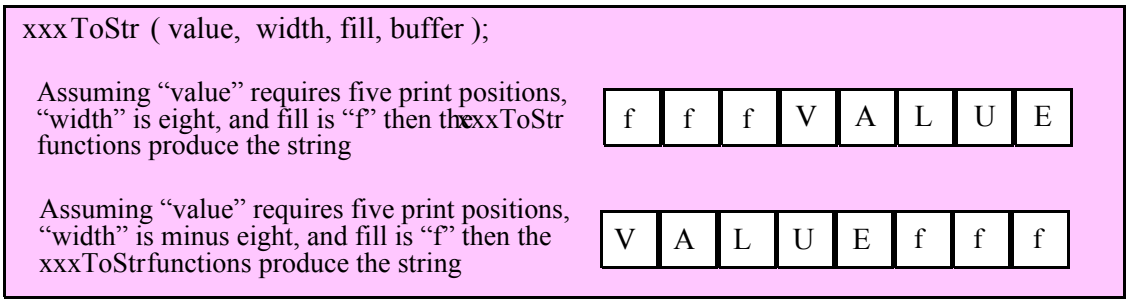
The hexadecimal numeric to string conversion routines are the general-purpose hexadecimal string conversion routines. There are two versions: one set of these routines store their string data into a preallocated string object (the *unadorned* versions), the other set (*adorned* with an "a_" prefix in their name) allocates storage for a new string on the heap and returns a pointer to that new string in the EAX register.

As for the buffer routines (e.g., conv.bToBuf and conv.h8ToBuf) there are two categories of routines based on whether the routines emit a mimimum length string or pad the string with leading zeros to the data type's *natural* size. The conv.h*X*ToStr routines emit the minimum number of hexadecimal digits in the string they create.

The conv.*X*ToStr (*X*= b, w, d, q, tb, or l) functions always create a fixed length string with an appropriate number of leading zeros (b=2 digits, w=4 digits, d=8 digits, q=16 digits, tb=20 digits, and l=32 digits). If the internal underscores flag contains true, then these functions will emit an underscore between each group of four hexadecimal digits.

The conv.h*X*ToStr functions (*X*=8, 16, 32,64, 80, or 128) let you specify a minimum field width and a padding character (because the conv.*X*ToBuf routines always emit fixed-length strings, there is no need to specify a minimum field width or padding character for those strings). The absolute value of the width parameter specifies the minimum string length for the conversion. The conversion will always produce a string at least abs(width) characters long. If the conversion would require more than abs(width) print positions, then the conversion will produce a larger string, as necessary.

If the string conversion requires fewer than abs(width) characters and the width parameter is a non-negative value, these routines right justify the value in the string and pad the remaining positions with the fill character. If the conversion requires fewer than abs(width) characters and the width parameter is a negative number, then these functions will left-justify the value in the output field and pad the end of the string with the fill character.



For the unadorned functions, the destination string's maximum length must be large enough to hold the full result (including any extra print positions needed beyond the value specified by abs(width)) or these functions will raise a string overflow exception.

If the internal underscore flag is true, then the 32-bit and larger hex to string conversion functions will emit an underscore between each set of four hexadecimal digits, starting from the least significant digit. This is true for both the conv.h*X*ToStr and conv.*X*ToStr routines. See the descriptions of the conv.setUnderscores and conv.getUnderscores functions for more details. Note that the hexadecimal to numeric conversion functions do not inject underscores into sequences of padding characters, only into the actual digits the conversion produces. This is true even if you specify a numeric character (such as '0') as the padding character.

8.3.4.1 Fixed-Length Numeric to Hexadecimal String Conversions

The functions in this category convert an 8-bit, 16-bit, 32-bit, 64-bit, 80-bit, or 128-bit numeric value to fixed-length strings (one hexadecimal digit for each four bits, including leading zeros). If the internal underscores flag contains true, then these functions will also insert an underscore between each group of four hexadecimal digits.

```
conv.bToStr( b:byte; dest:string );
```

This function converts the 8-bit value of the b parameter to a two-byte string containing b's hexadecimal representation. Because this conversion requires exactly two digits, the internal underscores flag setting does not affect the operation of this function. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

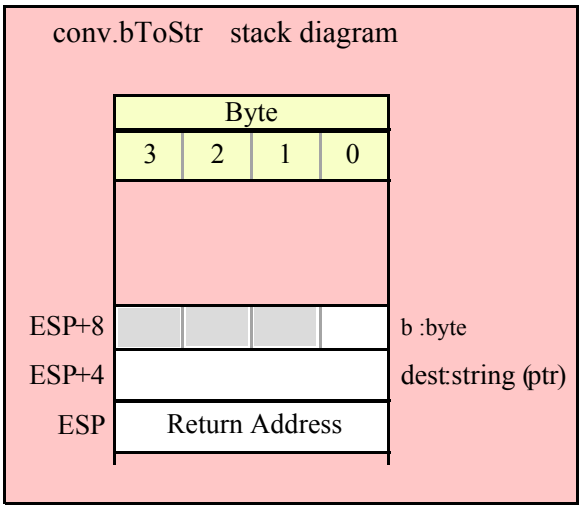
```
HLA high-level calling sequence examples:
```

```
// The following will push "byteVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.bToStr:

conv.bToStr( byteVariable, destStr );

// The following call will BH's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.bToStr:

conv.bToStr( bh, edx );

// The following pushes the constant and destStr and calls
// conv.bToStr:

conv.bToBuf( <constant>, destStr ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

```
HLA low-level calling sequence examples:

// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
push( destStr );
call conv.bToStr;

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
push( destStr );
call conv.bToStr;

// Passing a byte variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
push( destStr );
call conv.bToStr;


// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BL
push( edx );
call conv.bToStr;
```

```
// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
mov( bh, [esp] );
push( edx );
call conv.bToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.bToStr;
```

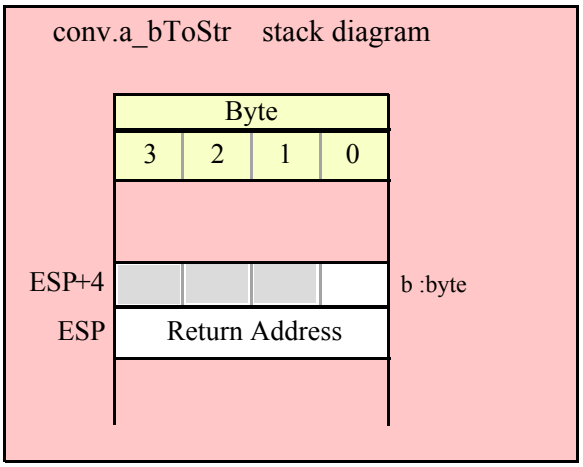


`conv.a_bToStr( b:byte ); @returns( "eax" );`

This function converts the value of the b parameter to a two-byte string containing b's hexadecimal representation. Because this conversion requires exactly two digits, the internal underscores flag setting does not affect the operation of this function. This function allocates storage for the string on the heap and returns a pointer to that string in the EAX register.

```
HLA high-level calling sequence examples:

// The following will push "byteVariable" and then call conv.a_bToStr:

conv.a_bToStr( byteVariable );
mov( eax, byteStr );

// The following call will BH's value onto the stack
// before calling conv.a_bToStr:

conv.a_bToStr( bh );
mov( eax, byteStr );

// The following pushes the constant and calls
// conv.a_bToStr:

conv.bToBuf( <constant> ); // <constant> must fit in 8 bits
mov( eax, byteStr );
```

```
HLA low-level calling sequence examples:

// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
call conv.a_bToStr;
mov( eax, destStr );

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
call conv.a_bToStr;
mov( eax, destStr );

// Passing a byte variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
call conv.a_bToStr;
mov( eax, destStr );

// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BL
call conv.a_bToStr;
mov( eax, byteStr );

// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
mov( bh, [esp] );
call conv.a_bToStr;
mov( eax, byteStr );

// Passing a constant:

pushd( <constant> );
call conv.a_bToStr;
mov( eax, byteStr );
```



conv.wToStr(w:word; dest:string);

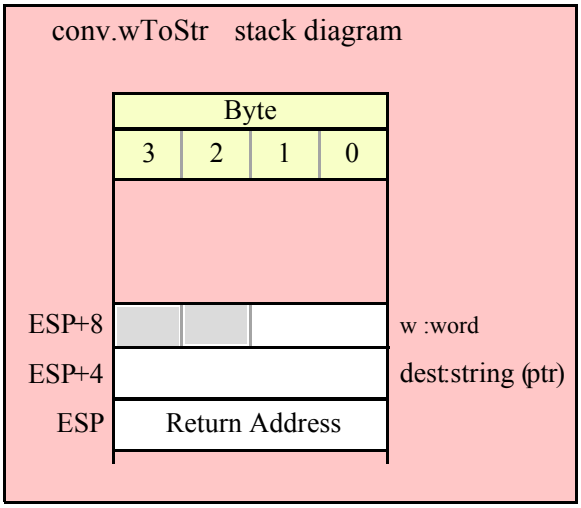
Converts the 16-bit value of the w parameter to a four-byte string that is the hexadecimal representation of this value. Because this conversions requires exactly four digits, the internal underscores flag setting does not affect the operation of this function. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

```
HLA high-level calling sequence examples:

// The following will push "wordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.wToStr:

conv.wToStr( wordVariable, destStr );

// The following call will BX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.wToStr:

conv.wToStr( bx, edx );

// The following pushes the constant and destStr and calls
// conv.wToStr:

conv.wToStr( <constant>, destStr ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

```
HLA low-level calling sequence examples:

// Passing a word variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
push( destStr );
call conv.wToStr;
```

```
// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

push( (type dword wordVariable));
push( destStr );
call conv.wToStr;

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:

pushw( 0 );
push( wordVariable );
push( destStr );
call conv.wToStr;


// Passing a pair of registers:
// BX = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BX
push( edx );
call conv.wToStr;


// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.wToStr;
```

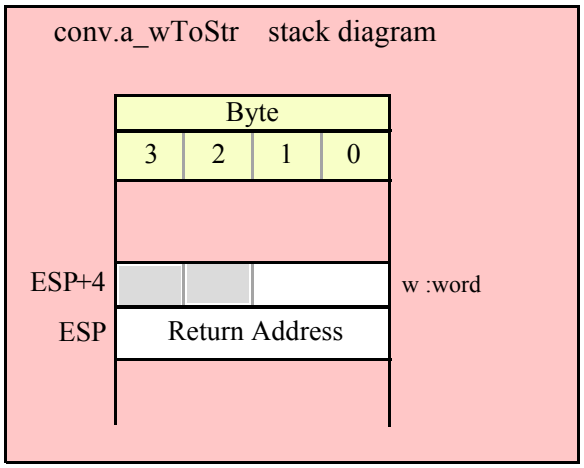


`conv.a_wToStr( w:word; dest:string );  @returns( "eax" );`

Converts the 16-bit value of the w parameter to a four-byte string that is the hexadecimal representation of this value. Because this conversions requires exactly four digits, the internal underscores flag setting does not affect the operation of this function. This function allocates storage for the string on the heap and returns a pointer to that string in the EAX register.

```
HLA high-level calling sequence examples:

// The following will push "wordVariable" and then call conv.wToStr:
```

```
conv.a_wToStr( wordVariable );
mov( eax, destStr );

// The following call will BX's value onto the stack
// before calling conv.a_wToStr:

conv.a_wToStr( bx );

// The following pushes the constant and calls
// conv.a_wToStr:

conv.a_wToStr( <constant>, destStr ); // <constant> must fit in 16 bits



HLA low-level calling sequence examples:

// Passing a word variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( wordVariable, eax );
push( eax );
call conv.a_wToStr;
mov( eax, destStr );

// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

push( (type dword wordVariable));
call conv.wToStr;
mov( eax, destStr );

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:

pushw( 0 );
push( wordVariable );
call conv.wToStr;
mov( eax, destStr );


// Passing a pair of registers:
// BX = value to print.

push( ebx );      // Pushes BX
call conv.wToStr;
mov( eax, wordStr );


// Passing a constant:

pushd( <constant> );
call conv.wToStr;
mov( eax, destStr );
```



conv.dToStr(d:dword; dest:string);

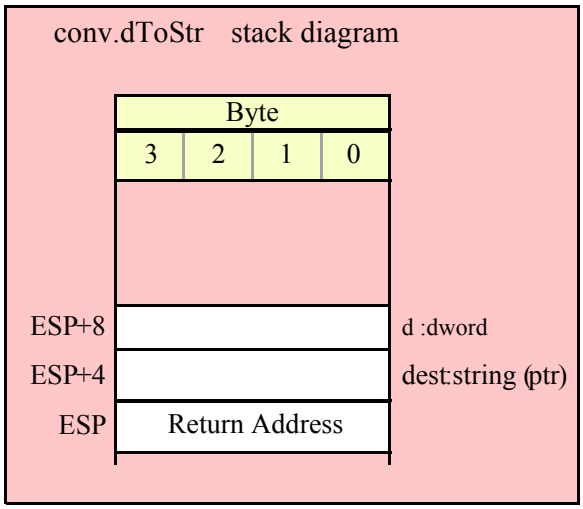
This function converts the 32-bit value of the d parameter to an eight- or nine-byte string that is the hexadecimal representation of this value. If d's value is greater than $FFFF and the underscores flag is true, then this function emits a nine-character string with an underscore between the fourth and fifth digits (counting from the least signficant digit). This function raises a string overflow error if the destination string is not large enough to hold the converted string.

```
HLA high-level calling sequence examples:

// The following will push "dwordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.dToStr:

conv.dToStr( dwordVariable, destStr );

// The following call will EBX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.dToStr:

conv.dToStr( ebx, edx );

// The following pushes the constant and destStr and calls
// conv.dToStr:

conv.dToStr( <constant>, destStr ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

```
HLA low-level calling sequence examples:


// Passing a dword variable and a buffer variable:

push( dwordVariable );
push( destStr );
call conv.dToStr;
```

```
// Passing a pair of registers:
// EBX = value to print, EDX = pointer to string object.

push( ebx );
push( edx );
call conv.dToStr;


// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.dToStr;
```

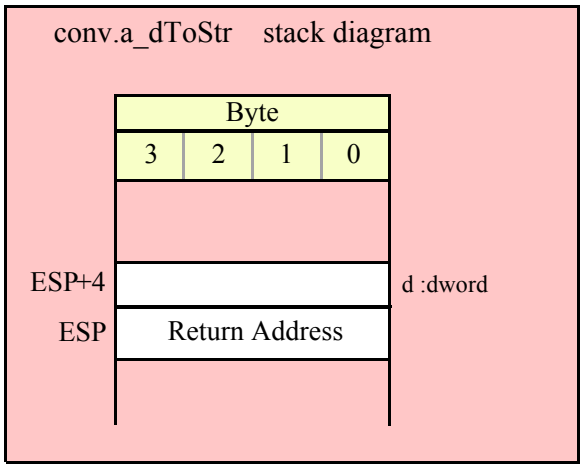


conv.a_dToStr(d:dword; dest:string); @returns("eax");

Converts the 32-bit value of the d parameter to an eight- or nine-byte string that is the hexadecimal representation of this value. If d's value is greater than $FFFF and the underscores flag is true, then this function emits a nine-character string with an underscore between the fourth and fifth digits (counting from the least signficant digit). This function allocates storage for the string on the heap and returns a pointer to that string in the EAX register.

```
HLA high-level calling sequence examples:

// The following will push "dwordVariable" and then call conv.a_dToStr:

conv.a_dToStr( dwordVariable );
mov( eax, destStr );

// The following call will push EBX's value onto the stack
// before calling conv.a_dToStr:

conv.a_dToStr( ebx );

// The following pushes the constant and calls
// conv.a_dToStr:

conv.a_dToStr( <constant>, destStr ); // <constant> must fit in 32 bits
```

```
HLA low-level calling sequence examples:

// Passing a dword variable:

push( dwordVariable );
call conv.a_dToStr;
mov( eax, destStr );


// Passing a register:
// EBX = value to print.

push( ebx );
call conv.a_dToStr;
mov( eax, dwordStr );


// Passing a constant:

pushd( <constant> );
call conv.a_dToStr;
mov( eax, destStr );
```

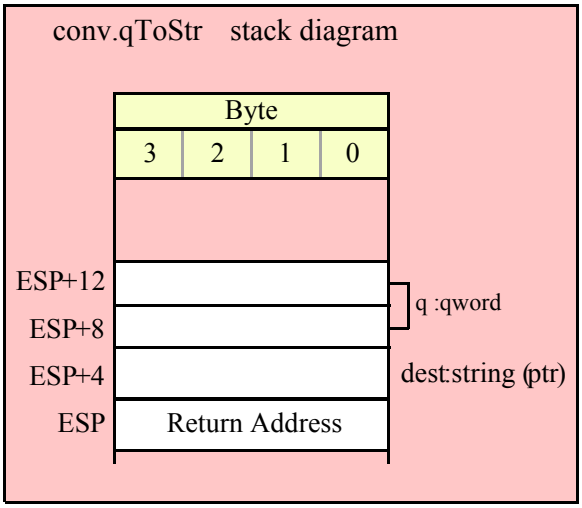


conv.qToStr(q:qword; dest:string);

Converts the 64-bit value of the q parameter to 16- or 19-byte string (based on the setting of the underscores flag) that is the hexadecimal representation of this value. If the internal underscores flag is true, then this function emits an underscore between each group of four hexadecimal digits (resulting in a 19-character string). This function raises a string overflow error if the destination string is not large enough to hold the converted string.

```
HLA high-level calling sequence examples:

// The following will push the value of "qwordVariable"
// and the value of the destStr string variable
// onto the stack and then call conv.qToStr:

conv.qToStr( qwordVariable, destStr );
```

```
// The following pushes the constant onto the stack along with
// the value held in the destStr variable and calls
// conv.qToStr:

conv.qToStr( <constant>, destStr ); // <constant> must fit in 64 bits


HLA low-level calling sequence examples:

// Passing a qword variable and a buffer variable:

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
push( destStr );
call conv.qToStr;


// Passing a constant:

pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
push( destStr );
call conv.qToStr;
```

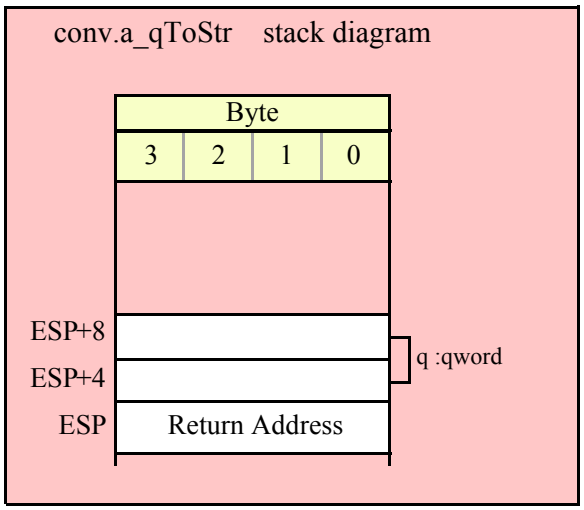


conv.a_qToStr(q:qword; dest:string); @returns("eax");

Converts the 64-bit value of the q parameter to 16- or 19-byte string (based on the setting of the underscores flag) that is the hexadecimal representation of this value. If the internal underscores flag is true, then this function emits an underscore between each group of four hexadecimal digits (resulting in a 19-character string). This function allocates storage for the string on the heap and returns a pointer to that string in the EAX register.

```
HLA high-level calling sequence examples:

// The following will push the value of "qwordVariable"
// onto the stack and then call conv.a_qToStr:

conv.a_qToStr( qwordVariable );
mov( eax, destStr );
```

```
// The following pushes the constant onto the stack and calls
// conv.a_qToStr:

conv.a_qToStr( <constant> ); // <constant> must fit in 64 bits
mov( eax, destStr );


HLA low-level calling sequence examples:

// Passing a qword variable:

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
call conv.a_qToStr;
mov( eax, destStr );


// Passing a constant:

pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
call conv.a_qToStr;
mov( eax, destStr );
```

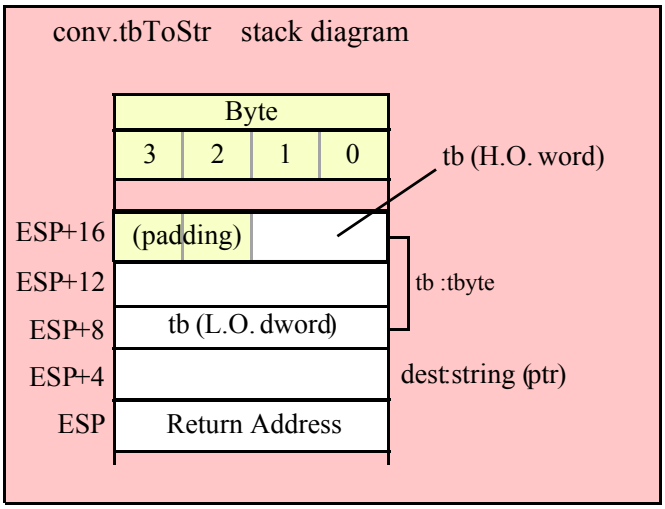


conv.tbToStr(tb:tbyte; dest:string);

Converts the 80-bit value of the d parameter to 20- or 24-byte string (based on the setting of the underscores flag) that is the hexadecimal representation of this value. If the internal underscores flag is true, then this function emits an underscore between each group of four hexadecimal digits (resulting in a 24-character string). This function raises a string overflow error if the destination string is not large enough to hold the converted string.

```
HLA high-level calling sequence examples:

// The following will push the value of "tbyteVariable"
// and the value of destStr onto the stack
// and then call conv.tbToStr:

conv.tbToStr( tbyteVariable, destStr );
```

```
HLA low-level calling sequence examples:

// Passing a tbyte variable:

pushw( 0 ); // Must pad parameter to 12 bytes
push( (type word tbyteVariable[8]));   // H.O. word first
  push( (type dword tbyteVariable[4]));
  push( (type dword qwordVariable[0]));// L.O. dword last
push( destStr );
call conv.tbToStr;

// Passing a constant:

pushd( <constant> >> 64 ); // Push H.O. word as dword, first
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
push( destStr );
call conv.tbToStr;
```

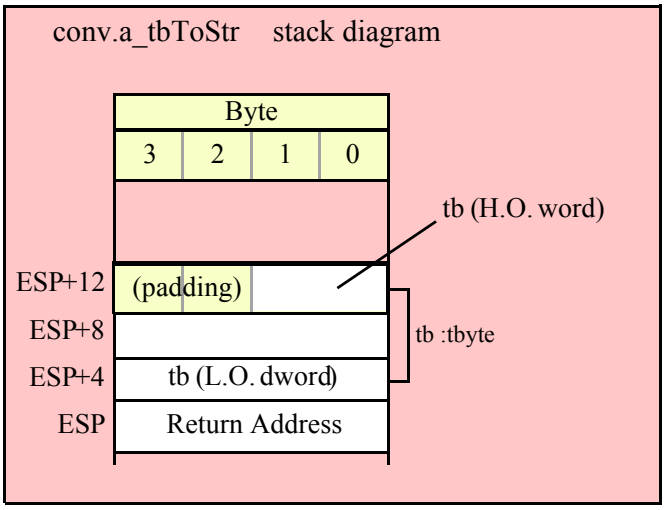


`conv.a_tbToStr( tb:tbyte; dest:string ); @returns( "eax" );`

Converts the 80-bit value of the d parameter to 20- or 24-byte string (based on the setting of the underscores flag) that is the hexadecimal representation of this value. If the internal underscores flag is true, then this function emits an underscore between each group of four hexadecimal digits (resulting in a 24-character string). This function allocates storage for the string on the heap and returns a pointer to that string in the EAX register.

```
HLA high-level calling sequence examples:

// The following will push the value of "tbyteVariable"
// onto the stack and then call conv.a_tbToStr:

conv.a_tbToStr( tbyteVariable );
mov( eax, destStr );


HLA low-level calling sequence examples:

// Passing a tbyte variable:

pushw( 0 ); // Must pad parameter to 12 bytes
```

```
    push( (type word tbyteVariable[8]));   // H.O. word first
      push( (type dword tbyteVariable[4]));
      push( (type dword qwordVariable[0]));// L.O. dword last
    call conv.a_tbToStr;
    mov( eax, destStr );

    // Passing a constant:

    pushd( <constant> >> 64 ); // Push H.O. word as dword, first
    pushd( (<constant> >> 32) & $FFFF_FFFF );
    pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
    call conv.a_tbToStr;
    mov( eax, destStr );
```

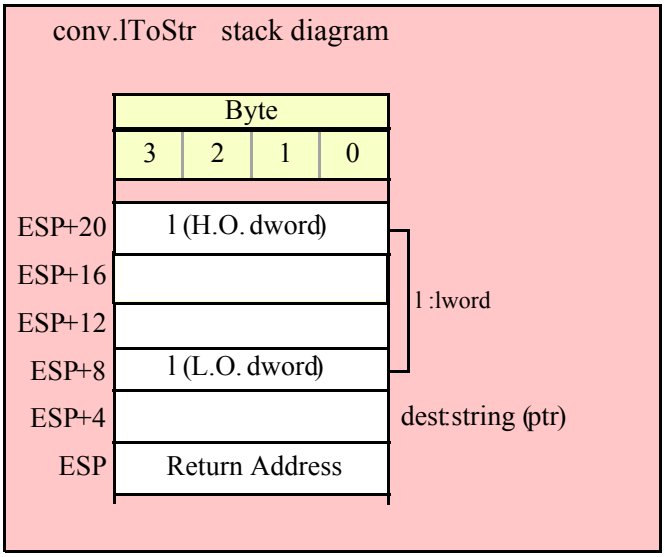


conv.lToStr(l:lword; dest:string);

Converts the 128-bit value of the l parameter to 32- or 19-byte string (based on the setting of the underscores flag) that is the hexadecimal representation of this value. If the internal underscores flag is true, then this function emits an underscore between each group of four hexadecimal digits (resulting in a 39-character string). This function raises a string overflow error if the destination string is not large enough to hold the converted string.

```
    HLA high-level calling sequence examples:

    // The following will push the value of "lwordVariable"
    // and destStr onto the stack, and then call conv.lToStr:

    conv.lToStr( lwordVariable, destStr );


    // The following pushes the constant onto the stack and calls
    // conv.lToStr:

    conv.lToStr( <constant>, edx ); // EDX contains string pointer value.


    HLA low-level calling sequence examples:

    // Passing an lword variable:

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

```
    push( (type dword lwordVariable[4]));
    push( (type dword lwordVariable[0]));// L.O. dword last
  push( destStr );
  call conv.lToStr;


  // Passing a constant:

  pushd( <constant> >> 96 );// Push H.O. dword of constant first.
  pushd( (<constant> >> 64)& $FFFF_FFFF );
  pushd( (<constant> >> 32)& $FFFF_FFFF );
  pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
  push( edx );// EDX contains string pointer value.
  call conv.lToStr;
```

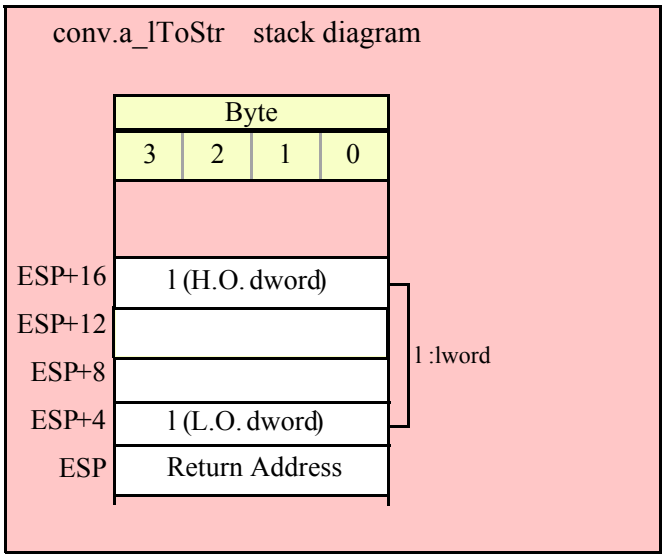


`conv.a_lToStr( l:lword; dest:string ); @returns( "eax" );`

Converts the 128-bit value of the l parameter to 32- or 19-byte string (based on the setting of the underscores flag) that is the hexadecimal representation of this value. If the internal underscores flag is true, then this function emits an underscore between each group of four hexadecimal digits (resulting in a 39-character string). This function allocates storage for the string on the heap and returns a pointer to that string in the EAX register.

```
  HLA high-level calling sequence examples:

    // The following will push the value of "lwordVariable"
    // onto the stack, and then call conv.a_lToStr:

    conv.a_lToStr( lwordVariable );
    mov( eax, destStr );


    // The following pushes the constant onto the stack and calls
    // conv.a_lToStr:

    conv.a_lToStr( <constant> );
    mov( eax, destStr );


  HLA low-level calling sequence examples:
```

```
// Passing an lword variable:

  push( (type dword lwordVariable[12]));// H.O. dword first
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
  call conv.a_lToStr;
  mov( eax, destStr );


// Passing a constant:

  pushd( <constant> >> 96 );// Push H.O. dword of constant first.
  pushd( (<constant> >> 64)& $FFFF_FFFF );
  pushd( (<constant> >> 32)& $FFFF_FFFF );
  pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
  call conv.a_lToStr;
  mov( eax, destStr );
```



8.3.4.2 Variable-Length Numeric to Hexadecimal String Conversions

The functions in this category convert a numeric value (8, 16, 32, 64, 80, or 128 bits) to a variable-length hexadecimal string (with no leading zeros). The string will contain the minimum number of digits needed to represent the value (plus underscores between each group of four digits if the internal underscores flag contains true).

`procedure conv.h8ToStr ( b:byte; width:int32; fill:char; buffer:string );`

Converts the 8-bit value of the b parameter to a one or two-byte string that is the hexadecimal representation of this value. Because 8-bit hexadecimal conversions require no more than two digits, the internal underscores flag setting does not affect the operation of this function. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

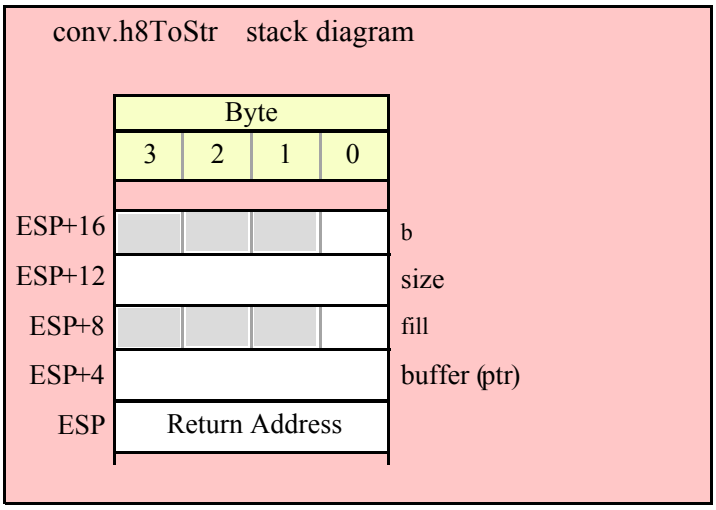
```
HLA high-level calling sequence examples:

// The following will push "byteVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.h8ToStr:

conv.h8ToStr( byteVariable, destStr );
```

```
// The following call will BH's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.h8ToStr:

conv.h8ToStr( bh, edx );

// The following pushes the constant and destStr and calls
// conv.h8ToStr:

conv.h8ToBuf( <constant>, destStr ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

```
HLA low-level calling sequence examples:

// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
push( destStr );
call conv.h8ToStr;

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
push( destStr );
call conv.h8ToStr;

// Passing a byte variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
push( destStr );
call conv.h8ToStr;


// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BL
push( edx );
call conv.h8ToStr;

// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
```

```
mov( bh, [esp] );
push( edx );
call conv.h8ToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.h8ToStr;
```

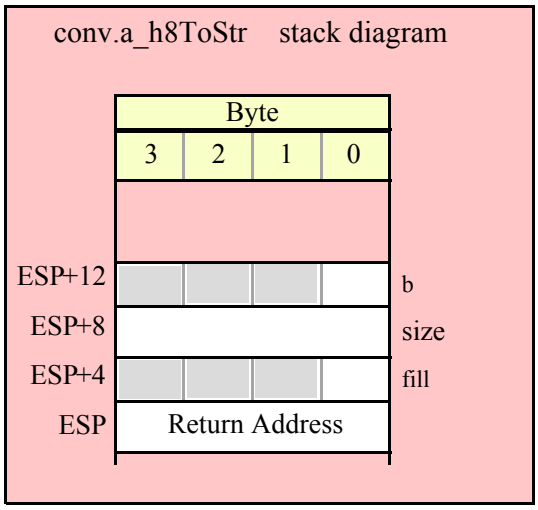


procedure conv.a_h8ToStr(b:byte; width:int32; fill:char);
@returns("eax");

Allocates an appropriate amount of string storage on the heap and then converts the 8-bit value of the b parameter to a 1..2 byte string that is the hexadecimal representation of this value. Returns a pointer to the newly allocated string in EAX. Because the number of digits is always two or less, the internal underscores flag does not affect the string this function produces.

```
HLA high-level calling sequence examples:

// The following will push "byteVariable" and then call conv.a_h8ToStr:

conv.a_h8ToStr( byteVariable );
mov( eax, byteStr );

// The following call will BH's value onto the stack
// before calling conv.a_h8ToStr:

conv.a_h8ToStr( bh );
mov( eax, byteStr );

// The following pushes the constant and calls
// conv.a_h8ToStr:

conv.a_h8ToStr( <constant> ); // <constant> must fit in 8 bits
mov( eax, byteStr );


HLA low-level calling sequence examples:

// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):
```

```
movzx( byteVariable, eax );
push( eax );
call conv.a_h8ToStr;
mov( eax, destStr );

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
call conv.a_h8ToStr;
mov( eax, destStr );

// Passing a byte variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
call conv.a_h8ToStr;
mov( eax, destStr );

// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BL
call conv.a_h8ToStr;
mov( eax, byteStr );

// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
mov( bh, [esp] );
call conv.a_h8ToStr;
mov( eax, byteStr );

// Passing a constant:

pushd( <constant> );
call conv.a_h8ToStr;
mov( eax, byteStr );
```



procedure conv.h16ToStr (w:word; width:int32; fill:char; buffer:string);

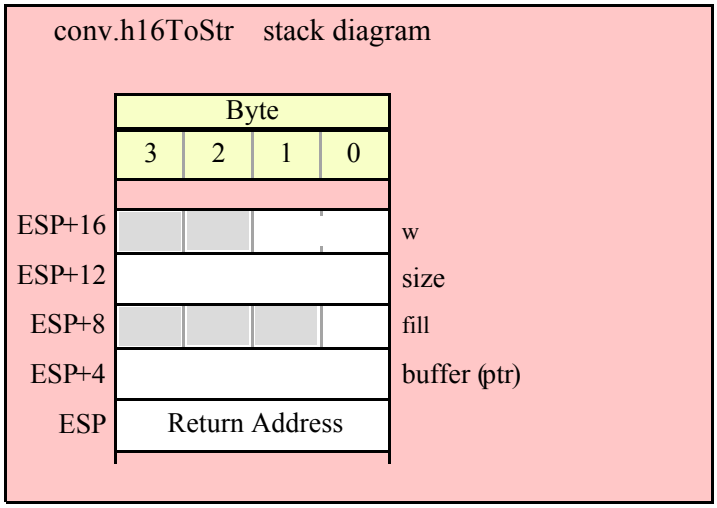
Converts the 16-bit value of the w parameter to a 1..4 byte string that is the hexadecimal representation of this value. Because 16-bit hexadecimal conversions require no more than four digits, the internal underscores flag setting does not affect the operation of this function. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

```
HLA high-level calling sequence examples:

// The following will push "wordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.h16ToStr:

conv.h16ToStr( wordVariable, destStr );

// The following call will BX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.h16ToStr:

conv.h16ToStr( bx, edx );

// The following pushes the constant and destStr and calls
// conv.h16ToStr:

conv.h16ToStr( <constant>, destStr ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

```
HLA low-level calling sequence examples:

// Passing a word variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
```

```
push( eax );
push( destStr );
call conv.h16ToStr;

// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

push( (type dword wordVariable));
push( destStr );
call conv.h16ToStr;

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:

pushw( 0 );
push( wordVariable );
push( destStr );
call conv.h16ToStr;


// Passing a pair of registers:
// BX = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BX
push( edx );
call conv.h16ToStr;


// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.h16ToStr;
```

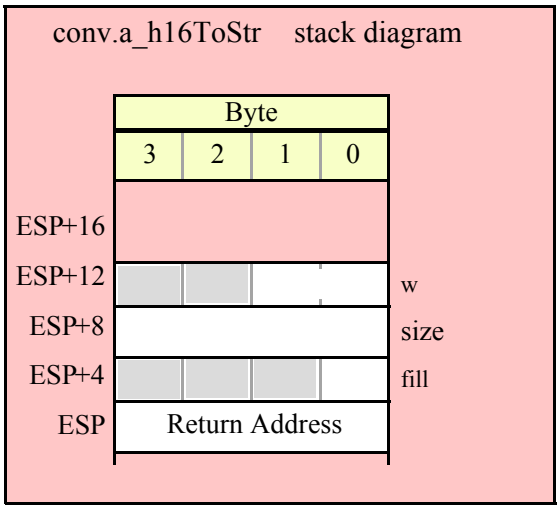


```
procedure conv.a_h16ToStr( w:word; width:int32; fill:char );
    @returns( "eax" );
```

Allocates an appropriate amount of string storage on the heap and then converts the 16-bit value of the w parameter to a 1..4 byte string that is the hexadecimal representation of this value. Returns a pointer to the newly allocated string in EAX. Because the number of digits is always four or less, the internal underscores flag does not affect the string this function produces.

```
HLA high-level calling sequence examples:

// The following will push "wordVariable" and call conv.a_h16ToStr:

conv.a_h16ToStr( wordVariable );
mov( eax, destStr );

// The following call will BX's value onto the stack
// before calling conv.a_h16ToStr:

conv.a_h16ToStr( bx );

// The following pushes the constant and calls
// conv.a_h16ToStr:

conv.a_h16ToStr( <const>, destStr ); // <const> must fit in 16 bits



HLA low-level calling sequence examples:

// Passing a word variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( wordVariable, eax );
push( eax );
call conv.a_h16ToStr;
mov( eax, destStr );

// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

push( (type dword wordVariable));
call conv.a_h16ToStr;
mov( eax, destStr );

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:

pushw( 0 );
push( wordVariable );
call conv.a_h16ToStr;
mov( eax, destStr );


// Passing a pair of registers:
// BX = value to print.

push( ebx );      // Pushes BX
call conv.a_h16ToStr;
mov( eax, wordStr );
```

```
// Passing a constant:

pushd( <constant> );
call conv.a_h16ToStr;
mov( eax, destStr );
```



procedure conv.h32ToStr (d:dword; width:int32; fill:char; buffer:string);

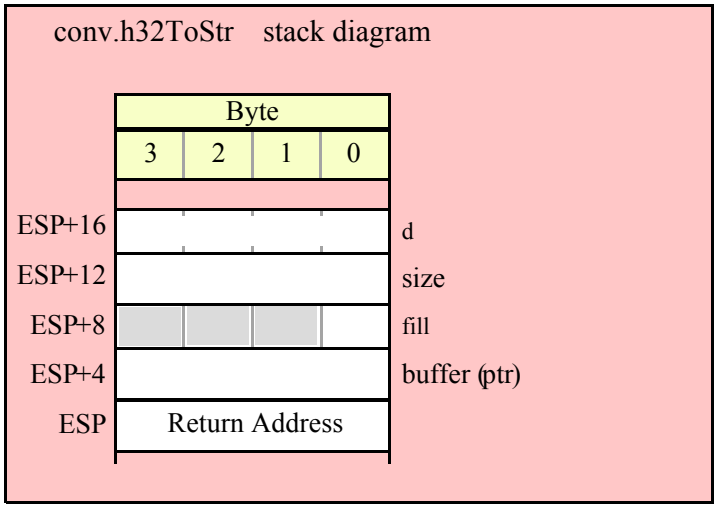
Converts the 32-bit value of the d parameter to a 1..8 byte string (1..9 bytes if the underscores flag is set) that is the hexadecimal representation of this value. If the 32-bit value is greater than $FFFF and the internal underscores flag is true, then this function will insert an underscore between the 4th and 5th digits of the string it produces. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

```
HLA high-level calling sequence examples:

// The following will push "dwordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.h32ToStr:

conv.h32ToStr( dwordVariable, destStr );

// The following call will push EBX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.h32ToStr:

conv.h32ToStr( ebx, edx );

// The following pushes the constant and destStr and calls
// conv.h32ToStr:

conv.h32ToStr( <constant>, destStr ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

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

```
// Passing a dword variable and a buffer variable:

push( dwordVariable );
push( destStr );
call conv.h32ToStr;


// Passing a pair of registers:
// EBX = value to print, EDX = pointer to string object.

push( ebx );
push( edx );
call conv.h32ToStr;


// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.h32ToStr;
```

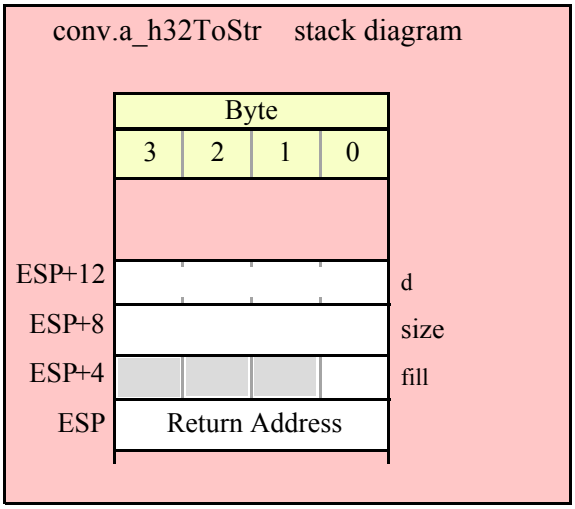


procedure conv.a_h32ToStr(d:dword; width:int32; fill:char);
@returns("eax");

Allocates an appropriate amount of string storage on the heap and then converts the 32-bit value of the d parameter to a 1..8 byte string (1..9 bytes if the underscores flag is set) that is the hexadecimal representation of this value. Returns a pointer to the newly allocated string in EAX. If the 32-bit value is greater than $FFFF and the internal underscores flag is true, then this function will insert underscores between each set of four digits to the left of the least significant digit in the string it produces.

```
HLA high-level calling sequence examples:

// The following will push "dwordVariable" and then call conv.a_h32ToStr:

conv.a_h32ToStr( dwordVariable );
mov( eax, destStr );

// The following call will push EBX's value onto the stack
```

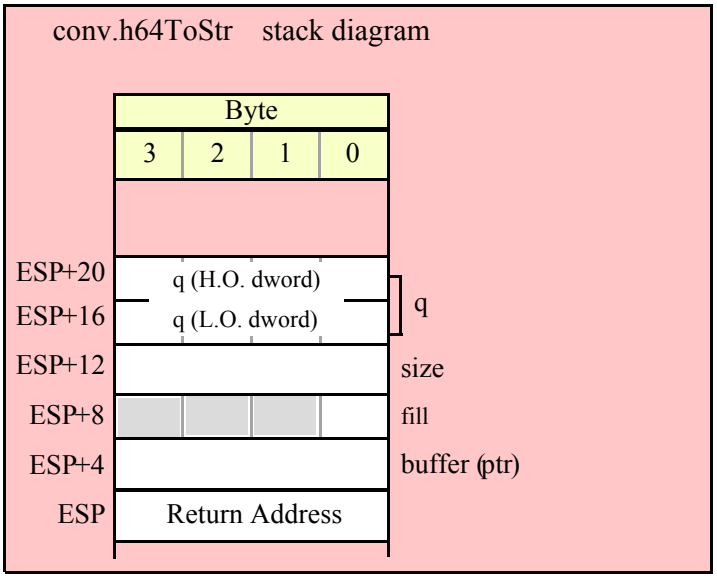
```
// before calling conv.a_h32ToStr:

conv.a_h32ToStr( ebx );

// The following pushes the constant and calls
// conv.a_h32ToStr:

conv.a_h32ToStr( <constant>, destStr ); // <constant> must fit in 32 bits



HLA low-level calling sequence examples:

// Passing a dword variable:

push( dwordVariable );
call conv.a_h32ToStr;
mov( eax, destStr );


// Passing a register:
// EBX = value to print.

push( ebx );
call conv.a_h32ToStr;
mov( eax, dwordStr );


// Passing a constant:

pushd( <constant> );
call conv.a_h32ToStr;
mov( eax, destStr );
```



procedure conv.h64ToStr (q:qword; width:int32; fill:char; buffer:string);

Converts the 64-bit value of the q parameter to a 1..16 byte string (1..19 bytes if the underscores flag is set) that is the hexadecimal representation of this value. If the 64-bit value is greater than $FFFF and the internal underscores flag is true, then this function will insert underscores between each set of four digits to the left of the least significant digit in the string it produces. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

```
HLA high-level calling sequence examples:

// The following will push the value of "qwordVariable"
// and the value of the destStr string variable
// onto the stack and then call conv.h64ToStr:

conv.h64ToStr( qwordVariable, destStr );


// The following pushes the constant onto the stack along with
// the value held in the destStr variable and calls
// conv.h64ToStr:

conv.h64ToStr( <constant>, destStr ); // <constant> must fit in 64 bits


HLA low-level calling sequence examples:

// Passing a qword variable and a buffer variable:

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
push( destStr );
call conv.h64ToStr;


// Passing a constant:

pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
push( destStr );
call conv.h64ToStr;
```



```
procedure conv.a_h64ToStr( q:qword;  width:int32; fill:char );
    @returns( "eax" );
```

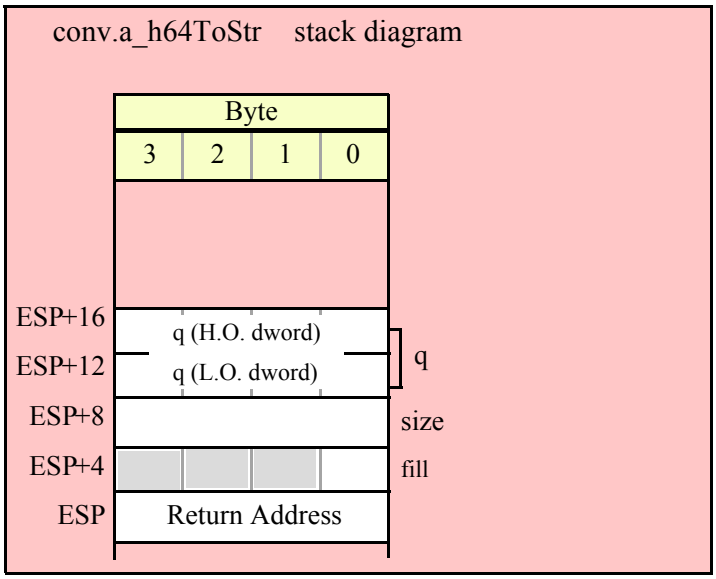
Allocates an appropriate amount of string storage on the heap and then converts the 64-bit value of the q parameter to a 1..16 byte string (1..19 bytes if the underscores flag is set) that is the hexadecimal representation of this value. Returns a pointer to the newly allocated string in EAX. If the 64-bit value is greater than $FFFF and the internal underscores flag is true, then this function will insert underscores between each set of four digits to the left of the least significant digit in the string it produces.

HLA high-level calling sequence examples:

```
// The following will push the value of "qwordVariable"
// onto the stack and then call conv.a_h64ToStr:

conv.a_h64ToStr( qwordVariable );
mov( eax, destStr );


// The following pushes the constant onto the stack and calls
// conv.a_h64ToStr:

conv.a_h64ToStr( <constant> ); // <constant> must fit in 64 bits
mov( eax, destStr );
```

HLA low-level calling sequence examples:

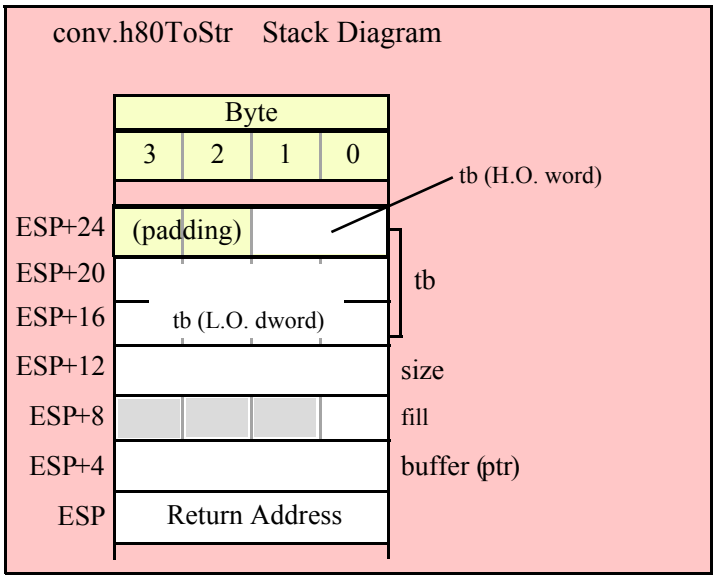
```
// Passing a qword variable:

                push( (type dword qwordVariable[4]));// H.O. dword first
                push( (type dword qwordVariable[0]));// L.O. dword last
call conv.a_h64ToStr;
mov( eax, destStr );


// Passing a constant:

pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
call conv.a_h64ToStr;
mov( eax, destStr );
```



procedure conv.h80ToStr(tb:tbyte; width:int32; fill:char; buffer:string);

Converts the 80-bit value of the tb parameter to a 1..20 byte string (1..24 bytes if the underscores flag is set) that is the hexadecimal representation of this value. If the 80-bit value is greater than $FFFF and the internal underscores flag is true, then this function will insert underscores between each set of four digits to the left of the least significant digit in the string it produces. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

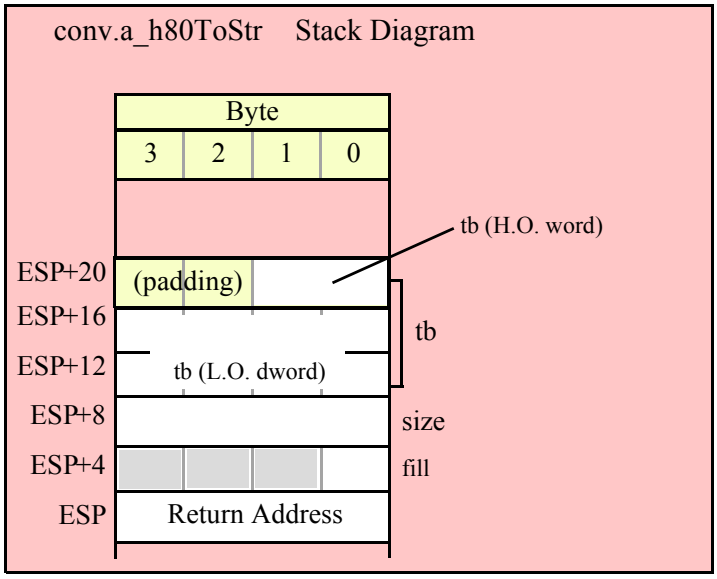
```
HLA high-level calling sequence examples:

// The following will push the value of "tbyteVariable"
// and the value of destStr onto the stack
// and then call conv.h80ToStr:

conv.h80ToStr( tbyteVariable, destStr );


HLA low-level calling sequence examples:

// Passing a tbyte variable:

pushw( 0 ); // Must pad parameter to 12 bytes
push( (type word tbyteVariable[8]));   // H.O. word first
  push( (type dword tbyteVariable[4]));
  push( (type dword qwordVariable[0]));// L.O. dword last
push( destStr );
call conv.h80ToStr;

// Passing a constant:

pushd( <constant> >> 64 ); // Push H.O. word as dword, first
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
push( destStr );
call conv.h80ToStr;
```



procedure conv.a_h80ToStr(tb:tbyte; width:int32; fill:char);
@returns("eax");

Allocates an appropriate amount of string storage on the heap and then converts the 80-bit value of the tb parameter to a 1..20 byte string (1..24 bytes if the underscores flag is set) that is the hexadecimal representation of this value. Returns a pointer to the newly allocated string in EAX. If the 80-bit value is greater than $FFFF and the internal underscores flag is true, then this function will insert underscores between each set of four digits to the left of the least significant digit in the string it produces.

```
HLA high-level calling sequence examples:

// The following will push the value of "tbyteVariable"
// onto the stack and then call conv.a_h80ToStr:

conv.a_h80ToStr( tbyteVariable );
mov( eax, destStr );


HLA low-level calling sequence examples:

// Passing a tbyte variable:

pushw( 0 ); // Must pad parameter to 12 bytes
push( (type word tbyteVariable[8]));   // H.O. word first
  push( (type dword tbyteVariable[4]));
  push( (type dword qwordVariable[0]));// L.O. dword last
call conv.a_h80ToStr;
mov( eax, destStr );

// Passing a constant:

pushd( <constant> >> 64 ); // Push H.O. word as dword, first
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
call conv.a_h80ToStr;
mov( eax, destStr );
```

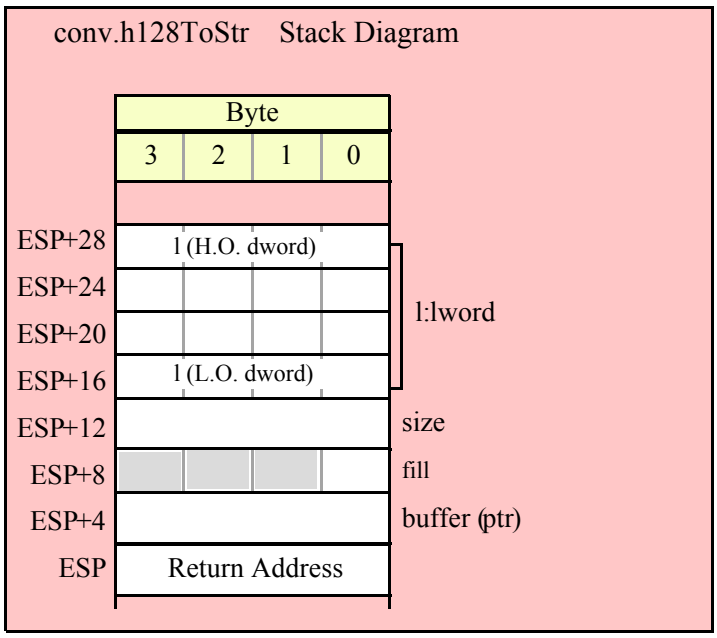


procedure conv.h128ToStr (l:lword; width:int32; fill:char; buffer:string);

Converts the 128-bit value of the l parameter to a 1..32 byte string (1..39 bytes if the underscores flag is set) that is the hexadecimal representation of this value. If the 128-bit value is greater than $FFFF and the internal underscores flag is true, then this function will insert underscores between each set of four digits to the left of the least significant digit in the string it produces. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

```
HLA high-level calling sequence examples:

// The following will push the value of "lwordVariable"
// and destStr onto the stack, and then call conv.h128ToStr:

conv.h128ToStr( lwordVariable, destStr );


// The following pushes the constant onto the stack and calls
// conv.h128ToStr:

conv.h128ToStr( <constant>, edx ); // EDX contains string pointer value.


HLA low-level calling sequence examples:

// Passing an lword variable:

  push( (type dword lwordVariable[12]));// H.O. dword first
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
push( destStr );
call conv.h128ToStr;


// Passing a constant:

pushd( <constant> >> 96 );// Push H.O. dword of constant first.
```

```
pushd( (<constant> >> 64)& $FFFF_FFFF );
pushd( (<constant> >> 32)& $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
push( edx );// EDX contains string pointer value.
call conv.h128ToStr;
```

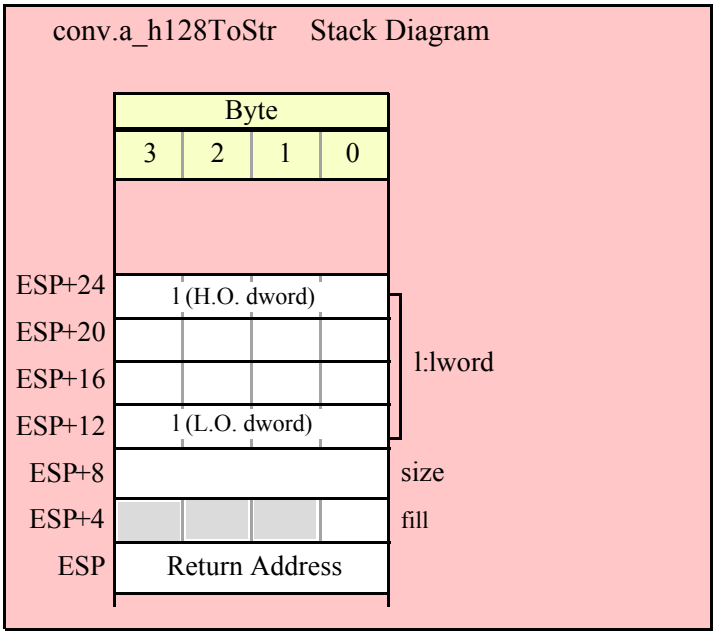


`conv.a_h128ToStr( l:lword;  width:int32; fill:char );`
`@returns( "eax" );`

Allocates an appropriate amount of string storage on the heap and then converts the 128-bit value of the l parameter to a 1..32 byte string (1..39 bytes if the underscores flag is set) that is the hexadecimal representation of this value. Returns a pointer to the newly allocated string in EAX. If the 128-bit value is greater than $FFFF and the internal underscores flag is true, then this function will insert underscores between each set of four digits to the left of the least significant digit in the string it produces.

```
HLA high-level calling sequence examples:

// The following will push the value of "lwordVariable"
// onto the stack, and then call conv.a_h128ToStr:

conv.a_h128ToStr( lwordVariable );
mov( eax, destStr );

// The following pushes the constant onto the stack and calls
// conv.a_h128ToStr:

conv.a_h128ToStr( <constant> );
mov( eax, destStr );


HLA low-level calling sequence examples:

// Passing an lword variable:

  push( (type dword lwordVariable[12]));// H.O. dword first
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
```

```
call conv.a_h128ToStr;
mov( eax, destStr );

// Passing a constant:

pushd( <constant> >> 96 );// Push H.O. dword of constant first.
pushd( (<constant> >> 64)& $FFFF_FFFF );
pushd( (<constant> >> 32)& $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
call conv.a_h128ToStr;
mov( eax, destStr );
```



8.3.5 Hexadecimal Buffer to Numeric Conversions

The hexadecimal buffer to numeric conversions ("ASCII to hexadecimal") convert a sequence of characters at some address in memory to numeric form.

These routines will skip over any leading underscore and delimiter characters (specified by the internal delimiters character set, see the discussion of conv.setDelimiters and conv.getDelimiters for details). These functions will convert all characters in the sequence until encountering a non-hexadecimal digit or underscore. If the first non-hex character is not the end of string or a delimiter character, these functions will raise a conversion exception. If the character is not a valid 7-bit ASCII character, these functions will raise an illegal character exception.

`procedure conv.atoh8( var buffer:var in esi ); @returns( "eax" );`

This function converts the hexadecimal character sequence beginning at character position held in ESI to an 8-bit numeric value. It raises an overflow exception if the value is outside the range $00..$FF. This function returns the result in AL, zero extended into EAX. This function returns ESI pointing at the first character after the sequence of hexadecimal characters.

```
HLA high-level calling sequence examples:

// The following will convert the hexadecimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in AL:
```

```
    conv.atoh8( [esi] );
    mov( al, hexNumericResult );


    // The following loads ESI with the address of
    // a sequence of hexadecimal characters (held in an HLA
    // string) and converts them to an 8-bit number:

    conv.atoh8( sourceStr ); // Loads "sourceStr" into ESI
    mov( al, byteVariable );


    HLA low-level calling sequence examples:

      // Same as first example above - ESI already contains
      // the address of the first character to convert:

      call conv.atoh8;
    mov( al, hexNumericResult );


    // Same as second example above

    static
      sourceStr :string := "12";
      .
      .
      .
  mov( sourceStr, esi );
    call conv.atoh8;
    mov( al, hex12 );
```

procedure conv.atoh16(var buffer:var in esi); @returns("eax");

This function converts the hexadecimal character sequence beginning at character position held in ESI to an 8-bit numeric value. It raises an overflow exception if the value is outside the range $0000..$FFFF. This function returns the result in AX, zero extended into EAX. This function returns ESI pointing at the first character after the sequence of hexadecimal characters.

```
    HLA high-level calling sequence examples:

    // The following will convert the hexadecimal characters in memory
    // at the address specified by [esi] into a numeric value
    // and return that value in AX:

    conv.atoh16( [esi] );
    mov( ax, hex16NumericResult );


    // The following loads ESI with the address of
    // a sequence of hexadecimal characters (held in an HLA
    // string) and converts them to a 16-bit number:

    conv.atoh16( sourceStr ); // Loads "sourceStr" into ESI
    mov( ax, wordVariable );


    HLA low-level calling sequence examples:
```

```
    // Same as first example above - ESI already contains
    // the address of the first character to convert:

    call conv.atoh16;
  mov( ax, hex16NumericResult );


  // Same as second example above

  static
    sourceStr :string := "12";
    .
    .
    .
mov( sourceStr, esi );
 call conv.atoh16;
 mov( ax, hex12 );
```

procedure conv.atoh32(var buffer:var in esi); @returns("eax");

This function converts the hexadecimal character sequence beginning at character position held in ESI to an 8-bit numeric value. It raises an overflow exception if the value is outside the range $0000_0000..$FFFF_FFFF. This function returns the result in EAX. This function returns ESI pointing at the first character after the sequence of hexadecimal characters.

```
  HLA high-level calling sequence examples:

  // The following will convert the hexadecimal characters in memory
  // at the address specified by [esi] into a numeric value
  // and return that value in EAX:

  conv.atoh32( [esi] );
  mov( eax, hex32NumericResult );


  // The following loads ESI with the address of
  // a sequence of hexadecimal characters (held in an HLA
  // string) and converts them to a 32-bit number:

  conv.atoh32( sourceStr ); // Loads "sourceStr" into ESI
  mov( eax, dwordVariable );


  HLA low-level calling sequence examples:

    // Same as first example above - ESI already contains
    // the address of the first character to convert:

    call conv.atoh32;
  mov( eax, hex32NumericResult );


  // Same as second example above

  static
    sourceStr :string := "12";
    .
    .
```

```
    .
  mov( sourceStr, esi );
   call conv.atoh32;
   mov( eax, hex12 );
```

procedure conv.atoh64(var buffer:var in esi); @returns("edx:eax");

This function converts the hexadecimal character sequence beginning at character position held in ESI to an 8-bit numeric value. It raises an overflow exception if the value is outside the range $0000_0000_0000_0000..$FFFF_FFFF_FFFF_FFFF. This function returns the result in EDX:EAX (H.O. double word in EDX). This function returns ESI pointing at the first character after the sequence of hexadecimal characters.

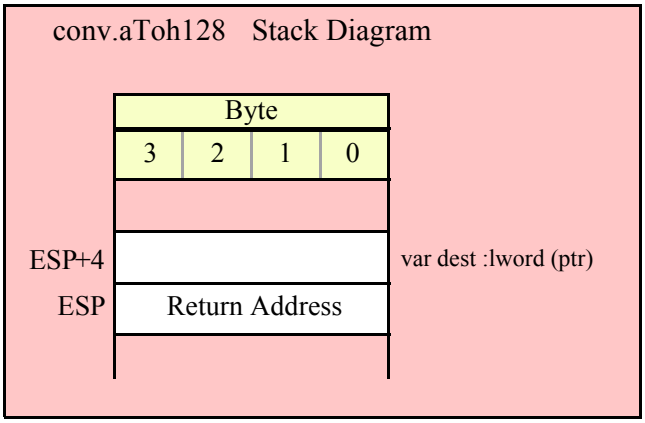
```
   HLA high-level calling sequence examples:

   // The following will convert the hexadecimal characters in memory
   // at the address specified by [esi] into a numeric value
   // and return that value in EDX:EAX:

   conv.atoh64( [esi] );
   mov( eax, (type dword hex64NumericResult[0]) );
   mov( edx, (type dword hex64NumericResult[4]) );


   // The following loads ESI with the address of
   // a sequence of hexadecimal characters (held in an HLA
   // string) and converts them to a 64-bit number:

   conv.atoh64( sourceStr ); // Loads "sourceStr" into ESI
   mov( eax, (type dword qwordVariable[0]) );
   mov( edx, (type dword qwordVariable[4]) );


   HLA low-level calling sequence examples:

     // Same as first example above - ESI already contains
     // the address of the first character to convert:

     call conv.atoh64;
   mov( eax, (type dword hex64NumericResult[0]) );
   mov( edx, (type dword hex64NumericResult[4]) );


   // Same as second example above

   static
     sourceStr :string := "12";
     .
     .
     .
  mov( sourceStr, esi );
   call conv.atoh64;
   mov( eax, (type dword qwordVariable[0]) );
   mov( edx, (type dword qwordVariable[4]) );
```

procedure conv.atoh128(var buffer:var in esi; var dest:lword);

This function converts the hexadecimal character sequence beginning at character position in the string to a 128-bit numeric value. It raises an overflow exception if the value is outside the range $0000_0000_0000_0000_0000_0000_0000_0000..$FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF. This function returns the result in the variable specified by the dest parameter.

```
HLA high-level calling sequence examples:

// The following will convert the hexadecimal characters in memory
// at the address specified by [esi] into a numeric value
// and stores that value in lwordDest (passed by reference):

conv.atoh128( [esi], lwordDest );


// The following loads ESI with the address of
// a sequence of hexadecimal characters (held in an HLA
// string) and converts them to a 128-bit number that it
// stores in lwordDest:

conv.atoh128( sourceStr, lwordDest ); // Loads "sourceStr" into ESI


HLA low-level calling sequence examples:

  // Option 1: lwordDest is a static object declared in a
  // HLA STATIC, READONLY, or STORAGE section:
  // As with the first example above, assume ESI already
  // contains the address of the string to convert:

  pushd( &lwordDest );// Pass address of lwordDest as reference parm.
  call conv.atoh128;

  // Option 2: lwordDest is a simple automatic variable (no indexing)
  // declared in a VAR section (or as a parameter). Assume that
  // no 32-bit registers can be disturbed by this code.
  // As with the first example above, assume ESI already
  // contains the address of the string to convert:

  push( ebp );
  add( @offset( lwordDest ), (type dword [esp]));
  call conv.atoh128;

  // Option 3: lwordDest is a complex addressing mode and at least
  // one 32-bit register is available for use by this code.
  // As with the first example above, assume ESI already
  // contains the address of the string to convert:

  lea( eax, lwordDest );// Assume EAX is the available register
  push( eax );
  call conv.atoh128;

// Same as second high-level example above. Assumes that
// lwordDest is a static object.

static
  sourceStr :string := "12";
  .
  .
  .
mov( sourceStr, esi );
```

```
pushd( &lwordDest );
call conv.atoh128;
```



8.3.6 Hexadecimal String to Numeric Conversions

This functions convert a string value, that contain the hexadecimal representation of a number, into the numeric form. These functions have two parameters: a string object and an index into that string. Numeric conversion begins at the zero-based character position specified by the index parameter. For example, the invocation

```
conv.strToh8( someStr, 5 );
```

begins the conversion starting with the sixth character (index 5) in someStr. These functions will raise an "index out of range" exception if the supplied index is greater than the size of the string the first parameter specifies. They will return a null pointer reference exception if the string parameter is NULL (they will return an illegal memory access exception if the first parameter is not a valid pointer and references unpaged memory).

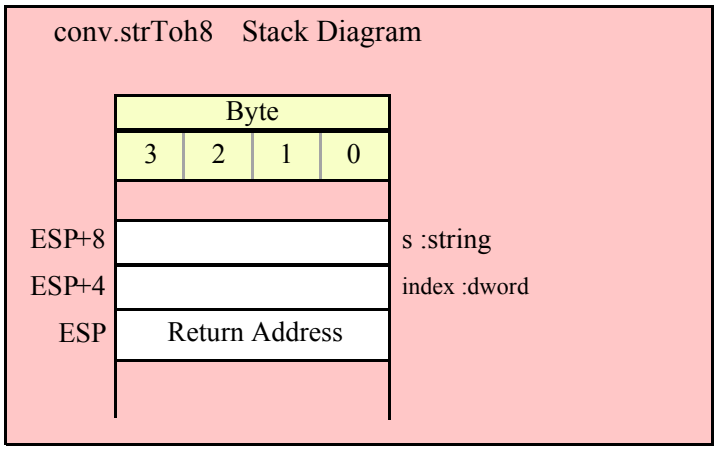
These routines will skip over any leading underscore and delimiter characters (specified by the internal delimiters character set, see the discussion of conv.setDelimiters and conv.getDelimiters for details). These functions will convert all characters in the sequence until encountering a non-hexadecimal digit or underscore. If the first non-hex character is not the end of string or a delimiter character, these functions will raise a conversion exception. If the character is not a valid 7-bit ASCII character, these functions will raise an illegal character exception.

```
procedure conv.strToh8( s:string; index:dword ); @returns( "eax" );
```

This function converts the hexadecimal character sequence beginning at the indexth character position in the string to an 8-bit numeric value. It raises an overflow exception if the value is outside the range $00..$FF. This function returns the result in AL, zero extended into EAX.

```
HLA high-level calling sequence examples:

// The following will convert the characters at the beginning
// of "hexValueStr" to numeric form:

conv.strToh8( hexValueStr, 0 );// Index=0 starts at beginning
mov( al, hexNumericResult );


// The following demonstrates using a non-zero index:

conv.strToh8( "abc12", 3 ); // "12" begins at offset 3
mov( al, hex12 );
```

```
HLA low-level calling sequence examples:

  push( hexValueStr );// Same as first example above
pushd( 0 );
call conv.strToh8;
mov( al, hexNumericResult );


// Same as second example above

static
  str12 :string := "abc12";
  .
  .
  .
push( str12 );// Note that str12 points at "abc12".
pushd( 3 );// Index to "12" in "abc12".
call conv.strToh8;
mov( al, hex12 );
```

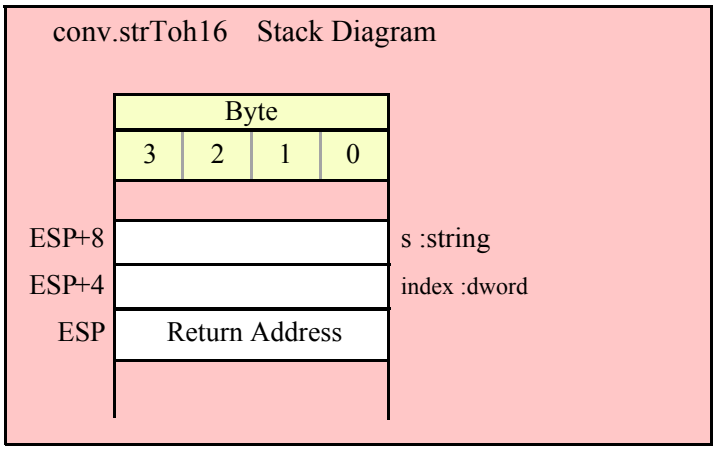


procedure conv.strToh16(s:string; index:dword); @returns("eax");

This function converts the hexadecimal character sequence beginning at the indexth character position in the string to a 16-bit numeric value. It raises an overflow exception if the value is outside the range $0000..$FFFF. This function returns the result in AX, zero extended into EAX.

```
HLA high-level calling sequence examples:

// The following will convert the characters at the beginning
// of "hexValueStr" to numeric form:

conv.strToh16( hexValueStr, 0 );// Index=0 starts at beginning
mov( ax, wordVar );


// The following demonstrates using a non-zero index:

conv.strToh16( "abc12FF", 3 ); // "12FF" begins at offset 3
mov( ax, wordVar );
```

```
HLA low-level calling sequence examples:

  push( hexValueStr );// Same as first example above
pushd( 0 );
call conv.strToh16;
mov( ax, wordVar );


// Same as second example above

static
  str12FF :string := "abc12FF";
  .
  .
  .
push( str12FF );// Note that str12FF points at "abc12FF".
pushd( 3 );// Index to "12FF" in "abc12FF".
call conv.strToh16;
mov( ax, wordVar );
```

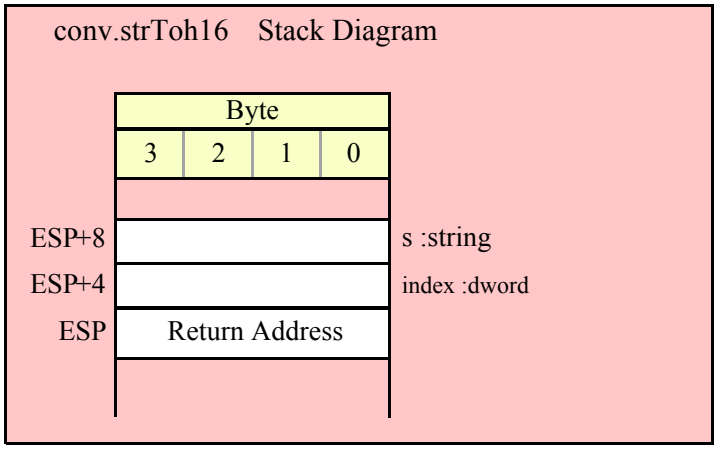


procedure conv.strToh32(s:string; index:dword); @returns("eax");

This function converts the hexadecimal character sequence beginning at the indexth character position in the string to a 32-bit numeric value. It raises an overflow exception if the value is outside the range $0000_0000..$FFFF_FFFF. This function returns the result in EAX.

```
HLA high-level calling sequence examples:

// The following will convert the characters at the beginning
// of "hexValueStr" to numeric form:

conv.strToh32( hexValueStr, 0 );// Index=0 starts at beginning
mov( eax, dwordVar );


// The following demonstrates using a non-zero index:

conv.strToh32( "abc12_FF00", 3 ); // "12_FF00" begins at offset 3
mov( eax, dwordVar );


HLA low-level calling sequence examples:

  push( hexValueStr );// Same as first example above
```

```
pushd( 0 );
call conv.strToh32;
mov( eax, dwordVar );


// Same as second example above

static
  str12FF00 :string := "abc12_FF00";
  .
  .
  .
push( str12FF00 );// Note that str12FF00 points at "abc12_FF00".
pushd( 3 ); // Index to "12_FF00" in "abc12_FF00".
call conv.strToh32;
mov( eax, dwordVar );// dwordVar now contains $12_FF00.
```

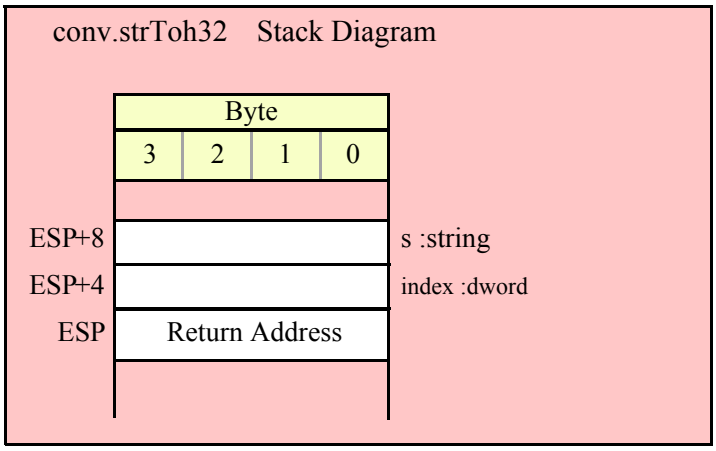


procedure conv.strToh64(s:string; index:dword); @returns("edx:eax");

This function converts the hexadecimal character sequence beginning at the indexth character position in the string to a 64-bit numeric value. It raises an overflow exception if the value is outside the range $0000_0000_0000_0000..$FFFF_FFFF_FFFF_FFFF. This function returns the result in EDX:EAX (H.O. double word in EDX).

```
HLA high-level calling sequence examples:

// The following will convert the characters at the beginning
// of "hexValueStr" to numeric form:

conv.strToh64( hexValueStr, 0 );// Index=0 starts at beginning
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );


// The following demonstrates using a non-zero index:

conv.strToh64( "abc12", 1 ); // "bc12" begins at offset 1
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );


HLA low-level calling sequence examples:
```

```
  push( hexValueStr );// Same as first example above
pushd( 0 );
call conv.strToh64;
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );


// Same as second example above

static
  strabc12 :string := "abc12";
  .
  .
  .
push( strabc12 );// Note that strabc12 points at "abc12".
pushd( 1 ); // Index to "bc12" in "abc12".
call conv.strToh64;
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );
```

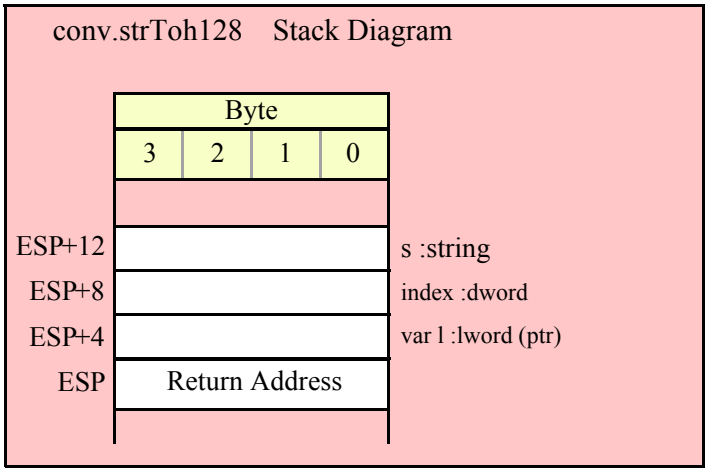


procedure conv.strToh128(s:string; index:dword; var dest:lword);

This function converts the hexadecimal character sequence beginning at the indexth character position in the string to a 128-bit numeric value. It raises an overflow exception if the value is outside the range $0000_0000_0000_0000_0000_0000_0000_0000..$FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF. This function returns the result in the variable specified by the dest parameter.

```
HLA high-level calling sequence examples:

// The following will convert the characters at the beginning
// of "hexValueStr" (index=0) to numeric form and store the
// 128-bit result into the 1wordDest variable:

conv.strToh128( hexValueStr, 0, 1wordDest );


// The following demonstrates using a non-zero index:

conv.strToh128( "abc1234567890abcdef", 3, 1wordDest );


HLA low-level calling sequence examples:
```

```
  //  Option #1: lwordDest is a STATIC/READONLY/STORAGE
  // variable:

  push( hexValueStr );// Same as first example above
pushd( 0 );
pushd( &lwordDest );
call conv.strToh128;

  // Option #2: lwordDest is not a static object and
  // a 32-bit register is available for use:

  push( hexValueStr );// Same as first example above
pushd( 0 );
lea( eax, lwordDest ); // Assuming EAX is available
push( eax );
call conv.strToh128;

  // Option #3: lwordDest is an automatic (var) object and
  // no 32-bit registers are available for use:

  push( hexValueStr );// Same as first example above
pushd( 0 );
push( ebp );
add( @offset( lwordDest ), (type dword [esp]) );
call conv.strToh128;

  // Option #4: lwordDest is a complex addressing mode object and
  // no 32-bit registers are available for use:

  push( hexValueStr );// Same as first example above
pushd( 0 );
sub( 4, esp );
push( eax );
lea( eax, lwordDest );
mov( eax, [esp+4] );
pop( eax );
call conv.strToh128;
```



8.4 Signed Integer Conversions

The integer conversion functions process signed integer values that are 8, 16, 32, 64, or 128 bits long. Functions in this category compute the output size (in print positions) of an integer, convert an integer to a sequence of characters, and convert a sequence of characters to an integer value.

8.4.1 Internal Functions

These functions are for internal use by the standard library. You should not call these functions in your own code. The internal functions in this category are conv._intToBuf32, conv._intToBuf32Size, conv._intToBuf64, conv._intToBuf64Size, conv._intToBuf128, conv._intToBuf128Size

8.4.2 Integer Size Calculations

These routines return the size, in screen print positions, it would take to print the signed integer passed in the specified parameter. They return their value in the EAX register (the value always fits in AL and AX, if you'd prefer to use these registes as the return value). The count includes room for a minus sign if the number is negative. Note that these routines include print positions required by underscores if you've enabled underscore output in values (see conv.setUnderscores and conv.getUnderscores for details).

procedure conv.i8Size(b:byte in al); @returns("eax");

This function computes the number of print positions required by the 8-bit signed integer passed in the AL register. Because the number of decimal positions is always three or less, the internal underscores flag does not affect the value this function returns. This function will always return a value in the range 1..4 (e.g., four positions for a value like "-128").

```
HLA high-level calling sequence examples:

conv.i8Size( byteVariable );
mov( eax, numSize );

conv.i8Size( <byte register> ); // al, ah, bl, bh, cl, ch, dl, dh
mov( eax, int8Size );

conv.i8Size( <constant> );      // Must fit into eight bits
mov( al, constantsSize );
```

Because conv.i8Size passes its input parameter in the AL register, any form of the high-level calling sequence except "conv.i8Size(al);" will automatically generate an instruction of the form "mov(<operand>,al);". Therefore, if possible, you should try to have the value whose size you wish to compute already sitting in the AL register and pass AL as the parameter to conv.i8Size.

```
HLA low-level calling sequence examples:

mov( byteVariable, al );
call conv.i8Size;
mov( eax, numSize );

mov( <byte register>, al ); // ah, bl, bh, cl, ch, dl, dh
call conv.i8Size;
mov( ax, wordVariable );

// Explicit Examples:

mov( bh, al );
call conv.i8Size;
mov( al, bhSize );
```

```
call conv.i8Size; // Assume value is already in AL
mov( al, alSize );


mov( 123, al );    // Example of computing the size of a constant
call conv.i8Size;
mov( eax, constsSize );
```

It might seem silly to compute the size of a constant as this last example is doing, as the constant's print width is known at compile time. Note, however, that this sequence could appear as part of a macro expansion and the literal constant "123" could actually be the result of expanding a macro parameter.

procedure conv.i16Size(w:word in ax); @returns("eax");

This function computes the number of print positions required by the 16-bit signed integer passed in the AX register. If the internal underscores flag is set and the integer value is greater than 999 (or less than -999) then this function will account for the underscores injected by the integer to string conversion routines. This function will always return a value in the range 1..6 if the underscores flag is not set (e.g., "-12345") or a value in the range 1…7 if the internal underscores flag is set (e.g., "-12_345").

```
HLA high-level calling sequence examples:

conv.i16Size( wordVariable );
mov( eax, numSize );

conv.i16Size( <word register> ); // ax, bx, cx, dx, bp, sp, si, di
mov( eax, int16Size );

conv.i16Size( <constant> );      // Must fit into 16 bits
mov( al, constantsSize );
```

Because conv.i16Size passes its input parameter in the AX register, any form of the high-level calling sequence except "conv.i16Size(ax);" will automatically generate an instruction of the form "mov*(<operand>*,ax);". Therefore, if possible, you should try to have the value whose size you wish to compute already sitting in the AX register and pass AX as the parameter to conv.i16Size.

```
HLA low-level calling sequence examples:

mov( wordVariable, ax );
call conv.i16Size;
mov( eax, numSize );

mov( <word register>, ax ); // bx, cx, dx, bp, sp, si, or di
call conv.i16Size;
mov( ax, wordVariable );

// Explicit Examples:

mov( bx, ax );
call conv.i16Size;
mov( al, bxSize );

call conv.i16Size; // Assume value is already in AX
mov( al, axSize );


mov( 12345, ax );    // Example of computing the size of a constant
call conv.i16Size;
mov( eax, constsSize );
```

See the comment at the end of conv.i8Size about passing constants to these functions.

procedure conv.i32Size(d:dword in eax); @returns("eax");

This function computes the number of print positions required by the 32-bit signed integer passed in the EAX register. If the internal underscores flag is set this function will include print positions for the underscores that the conversion routines will inject into the string. This function will always return a value in the range 1..11 if the underscores flag is not set (e.g., "-1000000000") or a value in the range 1…14 if the internal underscores flag is set (e.g., "-1_000_000_000").

```
HLA high-level calling sequence examples:

conv.i32Size( wordVariable );
mov( eax, numSize );

conv.i32Size( <dword register> ); // eax, ebx, ecx, edx,
mov( eax, int16Size )l// ebp, esp, esi, or edi

conv.i32Size( <constant> );      // Must fit into 32 bits
mov( al, constantsSize );
```

Because conv.i32Size passes its input parameter in the EAX register, any form of the high-level calling sequence except "conv.i32Size(eax);" will automatically generate an instruction of the form "mov*(<operand>*,eax);". Therefore, if possible, you should try to have the value whose size you wish to compute already sitting in the EAX register and pass EAX as the parameter to conv.i32Size.

```
HLA low-level calling sequence examples:

mov( dwordVariable, eax );
call conv.i32Size;
mov( eax, numSize );

mov( <dword register>, eax ); // ebx, ecx, edx,
call conv.i32Size;// ebp, esp, esi, or edi
mov( ax, wordVariable );

// Explicit Examples:

mov( ebx, eax );
call conv.i32Size;
mov( al, bxSize );

call conv.i32Size; // Assume value is already in AX
mov( al, axSize );


mov( 1234567890, eax );    // Example of computing
call conv.i32Size;// the size of a constant.
mov( eax, constsSize );
```

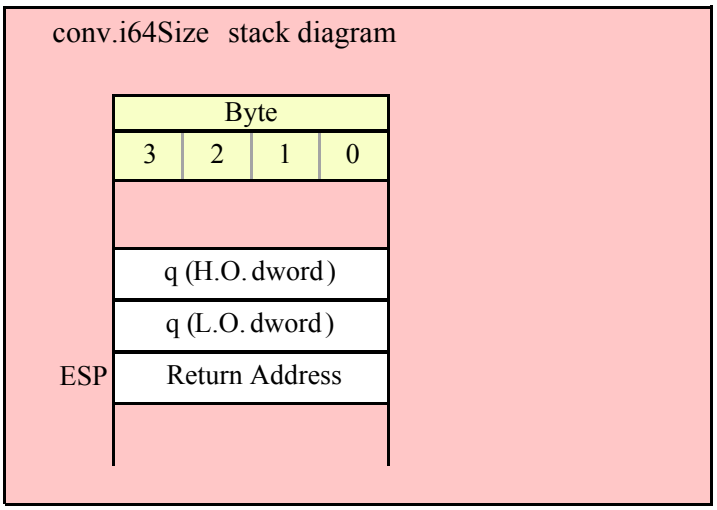
See the comment at the end of conv.i8Size about passing constants to these functions.

procedure conv.i64Size(q:qword); @returns("eax");

This function computes the number of print positions required by the 64-bit signed integer passed on the stack. If the internal underscores flag is set this function will include print positions for the underscores that the conversion routines will inject into the string. This function will always return a value in the range 1..20 if the underscores flag is not set (e.g., "-9223372036854775807") or a value in the range 1..26 if the internal underscores flag is set (e.g., "-9_223_372_036_854_775_808").

```
HLA high-level calling sequence examples:

conv.i64Size( qwordVariable );
mov( eax, numSize );

conv.i64Size( <constant> );      // Must fit into 64 bits
mov( al, constantsSize );


HLA low-level calling sequence examples:

push( (type dword qwordVariable[4]));// Push H.O. dword first
push( (type dword qwordVariable[0]));// Push L.O. dword second
call conv.i64Size;
mov( eax, numSize );

// Compute the size of a 64-bit constant:

pushd( 12345 >> 32 );// Push H.O. dword first
pushd( 12345 & $FFFF_FFFF );// Push L.O. dword second
call conv.i64Size;
mov( eax, constsSize );
```

See the comment at the end of conv.i8Size about passing constants to these functions. If you make a habit of explicitly passing 64-bit constants to this function, you might consider writing a macro to push the 64-bit constant for you (see the chapter on "Passing Parameters to Standard Library Routines" for more details).



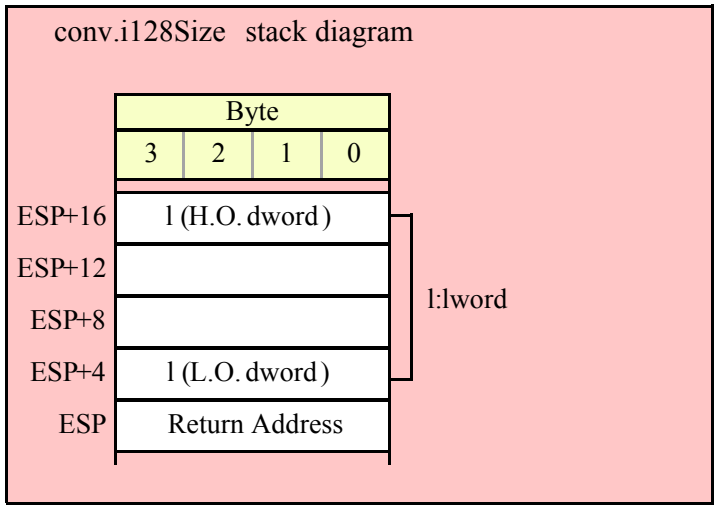
procedure conv.i128Size(l:lword); @returns("eax");

This function computes the number of print positions required by the 128-bit signed integer passed in the EAX register. If the internal underscores flag is set this function will include print positions for the underscores that the conversion routines will inject into the string. This function will always return a value in the range 1..40 if the underscores flag is not set (e.g., "-170141183460469231731687303715884105727") or a value in the range 1…52 if the internal underscores flag is set (e.g., "-170_141_183_460_469_231_731_687_303_715_884_105_728").

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

```
conv.i128Size( lwordVariable );
mov( eax, numSize );

conv.i128Size( <constant> );      // Must fit into 128 bits
mov( al, constantsSize );


HLA low-level calling sequence examples:

push( (type dword lwordVariable[12]));// Push H.O. dword first
push( (type dword lwordVariable[8]));
push( (type dword lwordVariable[4]));
push( (type dword lwordVariable[0]));// Push L.O. dword last
call conv.i64Size;
mov( eax, numSize );

// Compute the size of a 128-bit constant:

pushd( 12345 >> 96 );// Push H.O. dword first
pushd( (12345 >> 64) & $FFFF_FFFF );
pushd( (12345 >> 32 ) & $FFFF_FFFF );
pushd( 12345 & $FFFF_FFFF );// Push L.O. dword last
call conv.i128Size;
mov( eax, constsSize );
```

See the comment at the end of conv.i8Size about passing constants to these functions. If you make a habit of explicitly passing 128-bit constants to this function, you might consider writing a macro to push the 128-bit constant for you (see the chapter on "Passing Parameters to Standard Library Routines" for more details).



8.4.3 Signed Integer Numeric to Buffer Conversions

These routines convert the input parameter to a sequence of characters and store those characters starting at location [EDI]. They return EDI pointing at the first character beyond the converted string. Note that these functions do not zero terminate the string; if you want a zero-terminated string, then store a zero at the byte pointed at by EDI upon return from these functions.

If the internal underscores flag is set (see conv.getUnderscores and conv.setUnderscores for details), then these functions will insert an underscore between each group of three digits starting with the least significant digit.

procedure conv.i8ToBuf(i8 :int8 in al; var buf:var in edi);

This function converts the 8-bit signed integer passed in AL to a sequence of 1..4 characters. The string this function produces is always in the range -128..127. Note that because this string always contains three or fewer digits, the internal underscores flag setting does not affect this function's output.

```
HLA high-level calling sequence examples:

// The following will load "byteVariable" into AL and
// the address of "charArrayVariable" into EDI and then
// call conv.i8ToBuf:

conv.i8ToBuf( byteVariable, charArrayVariable );

// The following call will copy BH into AL and
// EDX into EDI prior to calling conv.i8ToBuf:

conv.i8ToBuf( bh, [edx] );

// The following just calls conv.i8ToBuf as AL and EDI
// already hold the parameter values:

conv.i8ToBuf( al, [edi] );

// The following loads the constant in AL and calls
// conv.i8ToBuf:

conv.i8ToBuf( <constant>, [edi] ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the AL and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. byte of EAX (i.e., AL) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AL and [EDI].

```
HLA low-level calling sequence examples:

// Passing a byte variable and a buffer variable:

mov( byteVariable, al );
lea( edi, charArrayVariable );
call conv.i8ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( byteVariable, al );
mov( &charArrayVariable, edi );
call conv.i8ToBuf;

// Passing a pair of registers (that are not
// AL and EDI):

mov( bh, al );
mov( edx, edi );
call conv.i8ToBuf;

// Passing a constant:

mov( <constant>, al );
```

```
call conv.i8ToBuf; // Assume EDI already contains buffer address.
```

procedure conv.i16ToBuf(i16 :int16 in ax; var buf:var in edi)

This function converts the 16-bit signed integer passed in AX to a sequence of 1..6 characters if the internal underscores flag is false, 1..7 characters if the underscores flag contains true. The string this function produces is always in the range -32768..32767. If the internal underscores flag contains true and the value is greater than 999 or less than -999, then this function emits an underscore between the third and fourth digits (from the right) in the string.

```
HLA high-level calling sequence examples:

// The following will load "wordVariable" into AX and
// the address of "charArrayVariable" into EDI and then
// call conv.i16ToBuf:

conv.i16ToBuf( wordVariable, charArrayVariable );

// The following call will copy BX into AX and
// EDX into EDI prior to calling conv.i16ToBuf:

conv.i16ToBuf( bx, [edx] );

// The following just calls conv.i16ToBuf as AX and EDI
// already hold the parameter values:

conv.i16ToBuf( ax, [edi] );

// The following loads the constant in AX and calls
// conv.i16ToBuf:

conv.i16ToBuf( <constant>, [edi] ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the AX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. word of EAX (i.e., AX) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AX and [EDI].

```
HLA low-level calling sequence examples:

// Passing a word variable and a buffer variable:

mov( wordVariable, ax );
lea( edi, charArrayVariable );
call conv.i16ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( wordVariable, ax );
mov( &charArrayVariable, edi );
call conv.i16ToBuf;

// Passing a pair of registers (that are not
// AX and EDI):

mov( bx, ax );
mov( edx, edi );
```

```
call conv.i16ToBuf;

// Passing a constant:

mov( <constant>, ax );
call conv.i16ToBuf; // Assume EDI already contains buffer address.
```

procedure conv.i32ToBuf(i32 :int32 in eax; var buf:var in edi)

This function converts the 32-bit signed integer passed in EAX to a sequence of 1..11 characters if the internal underscores flag is false, 1..14 characters if the underscores flag contains true. The string this function produces is always in the range -2147483648..2147483647. If the internal underscores flag contains true and the value is greater than 999 or less than -999, then this function emits an underscore between the third and fourth digits (from the right) in the string.

```
HLA high-level calling sequence examples:

// The following will load "dwordVariable" into EAX and
// the address of "charArrayVariable" into EDI and then
// call conv.i32ToBuf:

conv.i32ToBuf( dwordVariable, charArrayVariable );

// The following call will copy EBX into EAX and
// EDX into EDI prior to calling conv.i32ToBuf:

conv.i32ToBuf( ebx, [edx] );

// The following just calls conv.i32ToBuf as EAX and EDI
// already hold the parameter values:

conv.i32ToBuf( eax, [edi] );

// The following loads the constant in EAX and calls
// conv.i32ToBuf:

conv.i32ToBuf( <constant>, [edi] ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EAX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite EAX before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not EAX and [EDI].

```
HLA low-level calling sequence examples:

// Passing a dword variable and a buffer variable:

mov( dwordVariable, eax );
lea( edi, charArrayVariable );
call conv.i32ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( dwordVariable, eax );
mov( &charArrayVariable, edi );
call conv.i32ToBuf;
```

```
// Passing a pair of registers (that are not
// EAX and EDI):

mov( ebx, eax );
mov( edx, edi );
call conv.i32ToBuf;

// Passing a constant:

mov( <constant>, eax );
call conv.i32ToBuf; // Assume EDI already contains buffer address.
```

procedure conv.i64ToBuf(q :qword; var buf:var in edi)

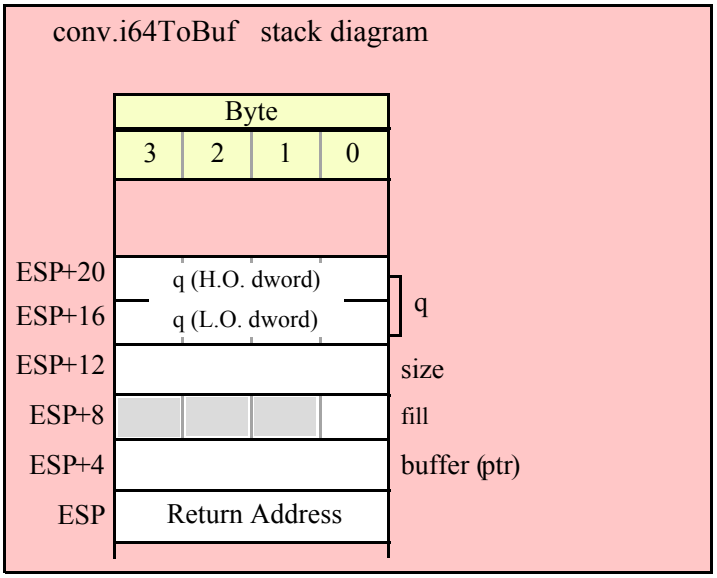
This function converts the 64-bit signed integer passed in q to a sequence of 1..20 characters if the internal underscores flag is false, 1..26 characters if the underscores flag contains true. The string this function produces is always in the range -9223372036854775808 .. 9223372036854775807. If the internal underscores flag contains true and the value is greater than 999 or less than -999, then this function emits an underscore between each group of three digits starting with the least significant digit.

```
HLA high-level calling sequence examples:

// The following will push the value of "qwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.i64ToBuf:

conv.i64ToBuf( qwordVariable, charArrayVariable );


// The following pushes the constant onto the stack and calls
// conv.i64ToBuf:

conv.i64ToBuf( <constant>, [edi] ); // <constant> must fit in 64 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

```
HLA low-level calling sequence examples:

// Passing a qword variable and a buffer variable:

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
lea( edi, charArrayVariable );
call conv.i64ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
mov( &charArrayVariable, edi );
call conv.i64ToBuf;

// Passing a constant:
```

```
pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
call conv.i64ToBuf; // Assume EDI already contains buffer address.
```



procedure conv.i128ToBuf(l :lword; var buf:var in edi)

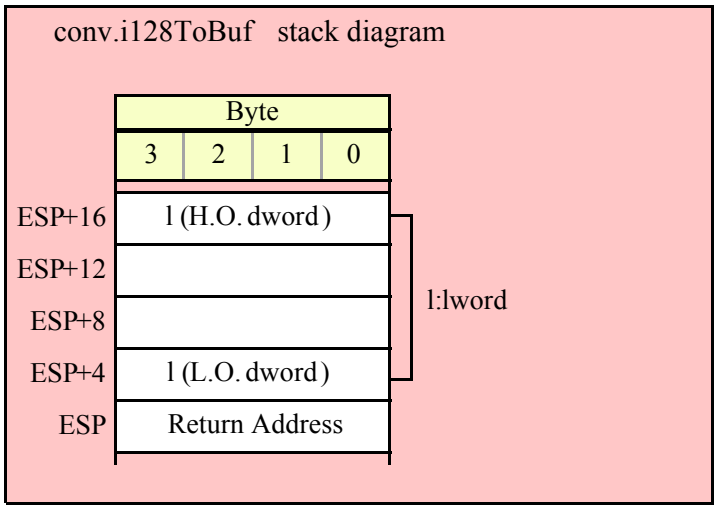
This function converts the 128-bit signed integer passed in l to a sequence of 1..40 characters if the internal underscores flag is false, 1..53 characters if the underscores flag contains true. The string this function produces is always in the range -170141183460469231731687303715884105728 .. 170141183460469231731687303715884105727. If the internal underscores flag contains true and the value is greater than 999 or less than -999, then this function emits an underscore between each group of three digits starting with the least significant digit.

```
HLA high-level calling sequence examples:

// The following will push the value of "lwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.i128ToBuf:

conv.i128ToBuf( lwordVariable, charArrayVariable );


// The following pushes the constant onto the stack and calls
// conv.i128ToBuf:

conv.i128ToBuf( <constant>, [edi] ); // <constant> must fit in 128 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

```
HLA low-level calling sequence examples:

// Passing an lword variable and a buffer variable:

  push( (type dword lwordVariable[12]));// H.O. dword first
```

```
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
lea( edi, charArrayVariable );
call conv.i128ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

  push( (type dword lwordVariable[12]));// H.O. dword first
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
mov( &charArrayVariable, edi );
call conv.i128ToBuf;

// Passing a constant:

pushd( <constant> >> 96 );// Push H.O. dword of constant first.
pushd( (<constant> >> 64)& $FFFF_FFFF );
pushd( (<constant> >> 32)& $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
call conv.i128ToBuf; // Assume EDI already contains buffer address.
```



8.4.4 Integer Numeric to String Conversions

These routines convert a signed integer value (8, 16, 32, 64, or 128 bits) to a string. The standard ("unadorned") functions store the string data into a string object that you pass as a parameter to the function. That string object must be preallocated and large enough to receive the string result (else a string overflow occurs). The "adorned" functions, whose names begin with "a_" automatically allocate storage on the heap, store the converted string into that heap object, and then return a pointer to the newly allocated string in the EAX register (it is the caller's responsibility to free the storage when it is no longer needed).

These functions let you specify a minimum field width and a fill character. If the number would require fewer than width print positions, the routines copy the fill character to the remaining positions in the destination string. If width is positive, the number is right justified in the string. If width is negative, the number is left justified in the string. If the string representation of the value requires more than width print positions, then these functions ignore the width and fill paramenters and use however many positions are necessary to properly display the value.

xxxToStr (value, width, fill, buffer);

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

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

Here are the maximum number of print positions these routines will produce for each data type before considering the minimum field width:

```
Underscores flag is false:
  8 bits:4 (-128..127)
  16 bits:6 (-32768..32767)
  32 bits:11 (-2147483648..2147483647)
  64 bits:20 (-9223372036854775808..9223372036854775807)
  128 bits:40 (-170141183460469231731687303715884105728 ..
                    170141183460469231731687303715884105728)

Underscores flag is true:
  8 bits:4 (-128..127)
  16 bits:7 (-32_768..32_767)
  32 bits:14 (-2_147_483_648..2_147_483_647)
  64 bits:26 (-9_223_372_036_854_775_808..9_223_372_036_854_775_807)
  128 bits:52 (-170_141_183_460_469_231_731_687_303_715_884_105_728 ..
                    170_141_183_460_469_231_731_687_303_715_884_105_728)
```

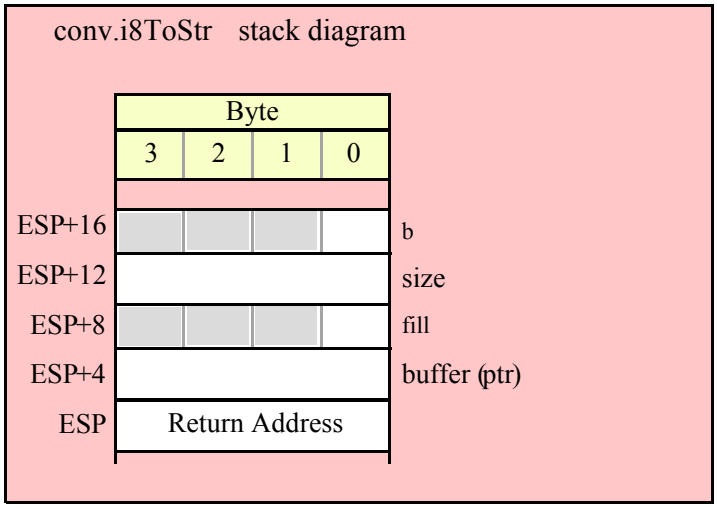
procedure conv.i8ToStr (b:int8; width:int32; fill:char; dest:string);

This function converts an 8-bit signed integer to the decimal string representation of that integer and stores the string in the preallocated string object specified by the dest paramenter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. Note that the internal underscores flag will not affect the output because 8-bit integers are always three digits or smaller.

```
HLA high-level calling sequence examples:

// The following will push "byteVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.i8ToStr:

conv.i8ToStr( byteVariable, destStr );

// The following call will BH's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.i8ToStr:

conv.i8ToStr( bh, edx );

// The following pushes the constant and destStr and calls
// conv.i8ToStr:

conv.bToBuf( <constant>, destStr ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

```
HLA low-level calling sequence examples:

// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
push( destStr );
call conv.i8ToStr;

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
push( destStr );
call conv.i8ToStr;

// Passing a byte variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
push( destStr );
call conv.i8ToStr;


// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BL
push( edx );
call conv.i8ToStr;

// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
mov( bh, [esp] );
push( edx );
call conv.i8ToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.i8ToStr;
```

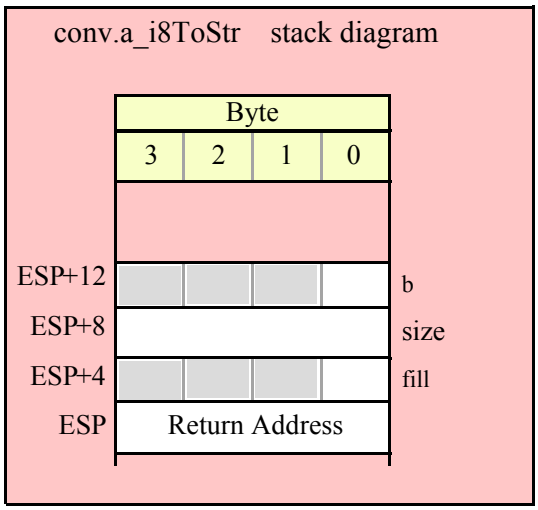



```
procedure conv.a_i8ToStr ( b:int8;  width:int32; fill:char );
    @returns( "eax" );
```

This function converts an 8-bit signed integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. Note that the internal underscores flag will not affect the output because 8-bit integers are always three digits or smaller.

```
HLA high-level calling sequence examples:

// The following will push "byteVariable" and then call conv.a_i8ToStr:

conv.a_i8ToStr( byteVariable );
mov( eax, byteStr );

// The following call will BH's value onto the stack
// before calling conv.a_i8ToStr:

conv.a_i8ToStr( bh );
mov( eax, byteStr );

// The following pushes the constant and calls
// conv.a_i8ToStr:

conv.a_i8ToStr( <constant> ); // <constant> must fit in 8 bits
mov( eax, byteStr );


HLA low-level calling sequence examples:

// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
call conv.a_i8ToStr;
mov( eax, destStr );
```

```
// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
call conv.a_i8ToStr;
mov( eax, destStr );

// Passing a byte variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
call conv.a_i8ToStr;
mov( eax, destStr );

// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BL
call conv.a_i8ToStr;
mov( eax, byteStr );

// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
mov( bh, [esp] );
call conv.a_i8ToStr;
mov( eax, byteStr );

// Passing a constant:

pushd( <constant> );
call conv.a_i8ToStr;
mov( eax, byteStr );
```



procedure conv.i16ToStr(w:int16; width:int32; fill:char; dest:string);

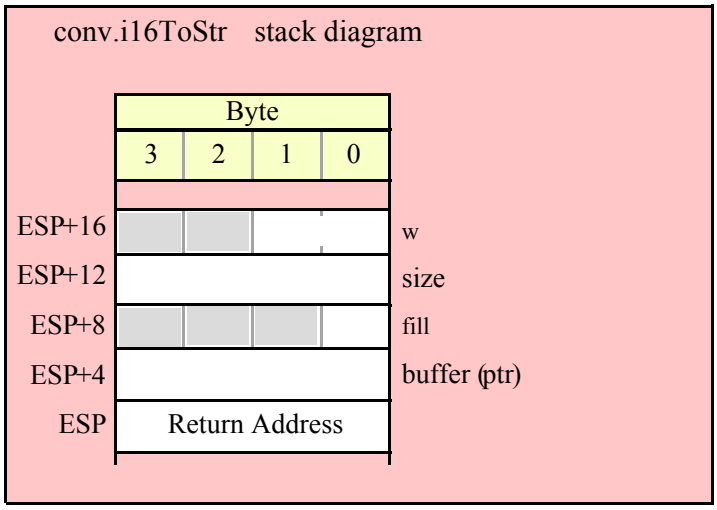
This function converts a 16-bit signed integer to its decimal string representation and stores the string in the preallocated string object specified by the dest paramenter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push "wordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.i16ToStr:

conv.i16ToStr( wordVariable, destStr );

// The following call will BX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.i16ToStr:

conv.i16ToStr( bx, edx );

// The following pushes the constant and destStr and calls
// conv.i16ToStr:

conv.i16ToStr( <constant>, destStr ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

```
HLA low-level calling sequence examples:

// Passing a word variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
push( destStr );
call conv.i16ToStr;

// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

push( (type dword wordVariable));
push( destStr );
call conv.i16ToStr;

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:
```

```
pushw( 0 );
push( wordVariable );
push( destStr );
call conv.i16ToStr;


// Passing a pair of registers:
// BX = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BX
push( edx );
call conv.i16ToStr;


// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.i16ToStr;
```

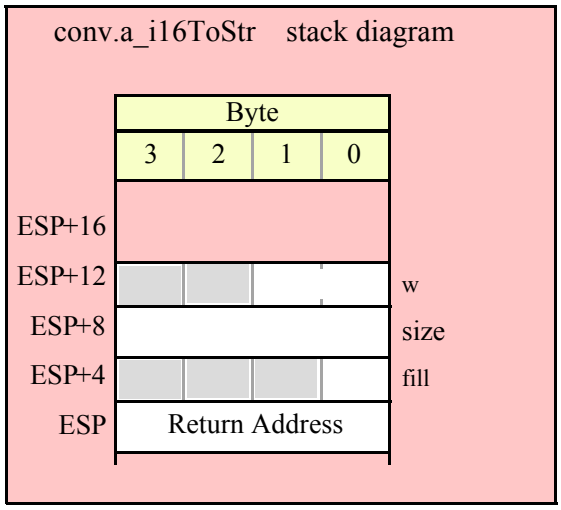


procedure conv.a_i16ToStr(w:int16; width:int32; fill:char);
@returns("eax");

This function converts a 16-bit signed integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push "wordVariable" and call conv.a_i16ToStr:

conv.a_i16ToStr( wordVariable );
mov( eax, destStr );

// The following call will BX's value onto the stack
```

```
// before calling conv.a_i16ToStr:

conv.a_i16ToStr( bx );

// The following pushes the constant and calls
// conv.a_i16ToStr:

conv.a_i16ToStr( <const>, destStr ); // <const> must fit in 16 bits



HLA low-level calling sequence examples:

// Passing a word variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( wordVariable, eax );
push( eax );
call conv.a_i16ToStr;
mov( eax, destStr );

// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

push( (type dword wordVariable));
call conv.a_i16ToStr;
mov( eax, destStr );

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:

pushw( 0 );
push( wordVariable );
call conv.a_i16ToStr;
mov( eax, destStr );


// Passing a pair of registers:
// BX = value to print.

push( ebx );      // Pushes BX
call conv.a_i16ToStr;
mov( eax, wordStr );


// Passing a constant:

pushd( <constant> );
call conv.a_i16ToStr;
mov( eax, destStr );
```



procedure conv.i32ToStr(d:int32; width:int32; fill:char; buffer:string);

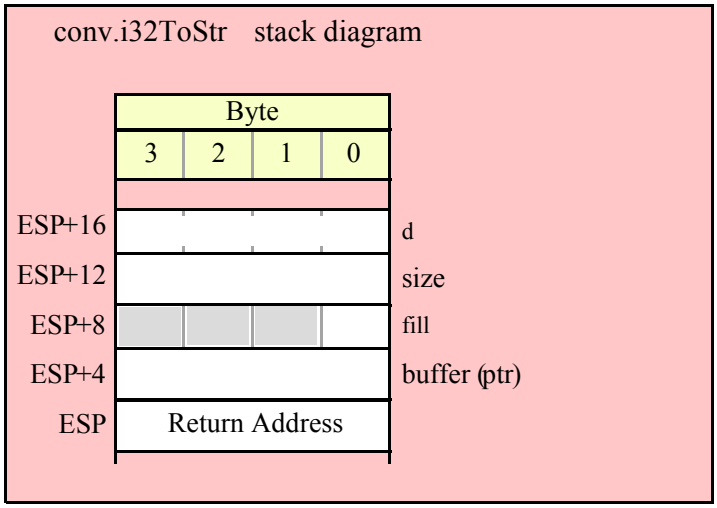
This function converts a 32-bit signed integer to its decimal string representation and stores the string in the preallocated string object specified by the buffer paramenter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push "dwordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.i32ToStr:

conv.i32ToStr( dwordVariable, destStr );

// The following call will push EBX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.i32ToStr:

conv.i32ToStr( ebx, edx );

// The following pushes the constant and destStr and calls
// conv.i32ToStr:

conv.i32ToStr( <constant>, destStr ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data. In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string data.

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

```
// Passing a dword variable and a buffer variable:

push( dwordVariable );
push( destStr );
call conv.i32ToStr;


// Passing a pair of registers:
// EBX = value to print, EDX = pointer to string object.

push( ebx );
push( edx );
call conv.i32ToStr;


// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.i32ToStr;
```



procedure conv.a_i32ToStr(d:int32; width:int32; fill:char);
@returns("eax");

This function converts a 32-bit signed integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push "dwordVariable" and then call conv.a_i32ToStr:

conv.a_i32ToStr( dwordVariable );
mov( eax, destStr );
```

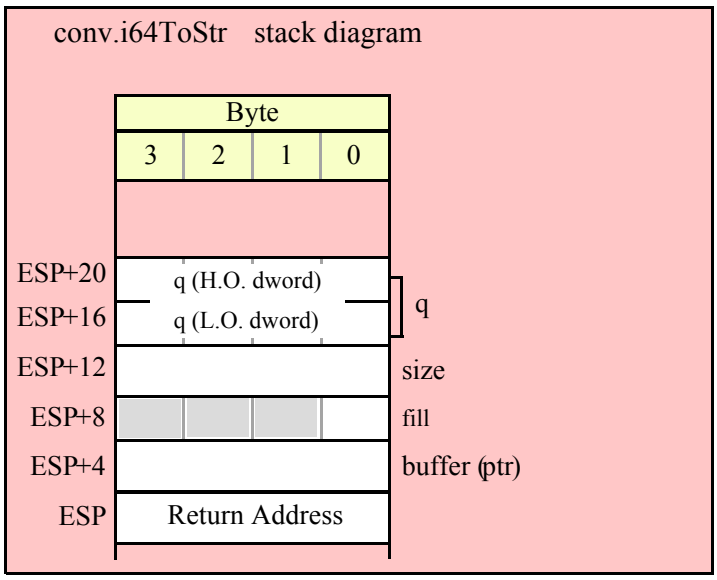
```
// The following call will push EBX's value onto the stack
// before calling conv.a_i32ToStr:

conv.a_i32ToStr( ebx );

// The following pushes the constant and calls
// conv.a_i32ToStr:

conv.a_i32ToStr( <constant>, destStr ); // <constant> must fit in 32 bits



HLA low-level calling sequence examples:

// Passing a dword variable:

push( dwordVariable );
call conv.a_i32ToStr;
mov( eax, destStr );


// Passing a register:
// EBX = value to print.

push( ebx );
call conv.a_i32ToStr;
mov( eax, dwordStr );


// Passing a constant:

pushd( <constant> );
call conv.a_i32ToStr;
mov( eax, destStr );
```

procedure conv.i64ToStr(q:qword; width:int32; fill:char; buffer:string);

This function converts a 64-bit signed integer to its decimal string representation and stores the string in the preallocated string object specified by the buffer paramenter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push the value of "qwordVariable"
// and the value of the destStr string variable
// onto the stack and then call conv.i64ToStr:

conv.i64ToStr( qwordVariable, destStr );


// The following pushes the constant onto the stack along with
// the value held in the destStr variable and calls
// conv.i64ToStr:
```

```
conv.i64ToStr( <constant>, destStr ); // <constant> must fit in 64 bits


HLA low-level calling sequence examples:

// Passing a qword variable and a buffer variable:

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
push( destStr );
call conv.i64ToStr;


// Passing a constant:

pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
push( destStr );
call conv.i64ToStr;
```

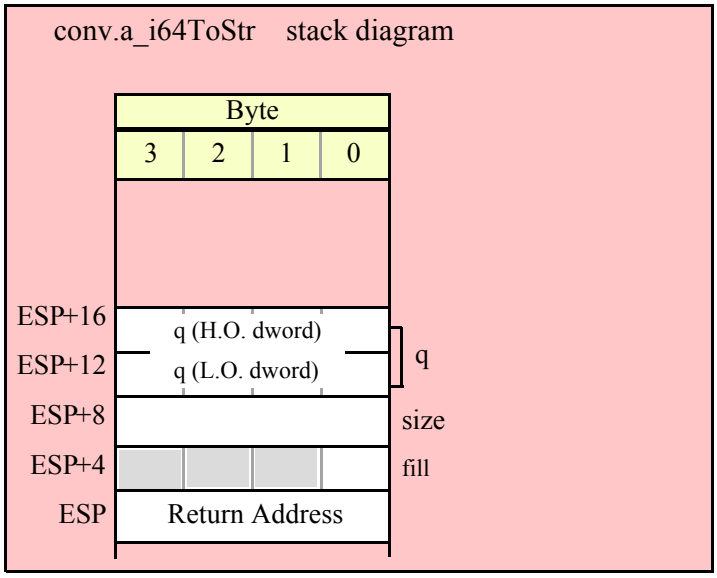


procedure conv.a_i64ToStr(q:qword; width:int32; fill:char);
@returns("eax");

This function converts a 64-bit signed integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push the value of "qwordVariable"
// onto the stack and then call conv.a_i64ToStr:

conv.a_i64ToStr( qwordVariable );
```

```
mov( eax, destStr );


// The following pushes the constant onto the stack and calls
// conv.a_i64ToStr:

conv.a_i64ToStr( <constant> ); // <constant> must fit in 64 bits
mov( eax, destStr );


HLA low-level calling sequence examples:

// Passing a qword variable:

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
call conv.a_i64ToStr;
mov( eax, destStr );


// Passing a constant:

pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
call conv.a_i64ToStr;
mov( eax, destStr );
```

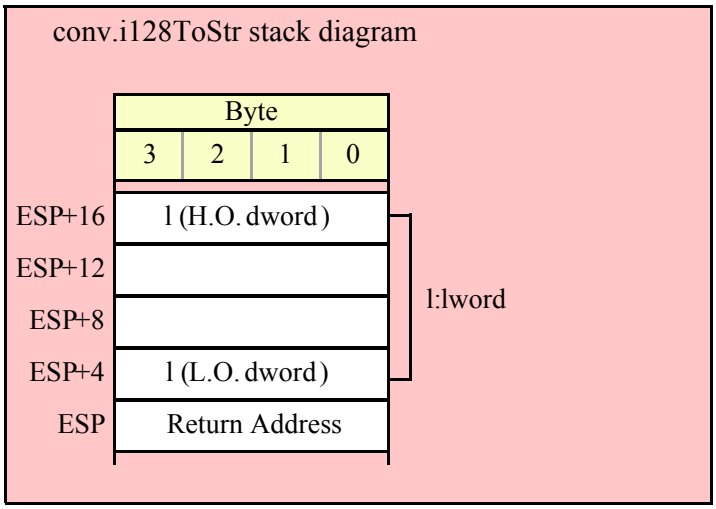


procedure conv.i128ToStr(l:lword; width:int32; fill:char; buffer:string);

This function converts a 128-bit signed integer to its decimal string representation and stores the string in the preallocated string object specified by the buffer paramenter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag

is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

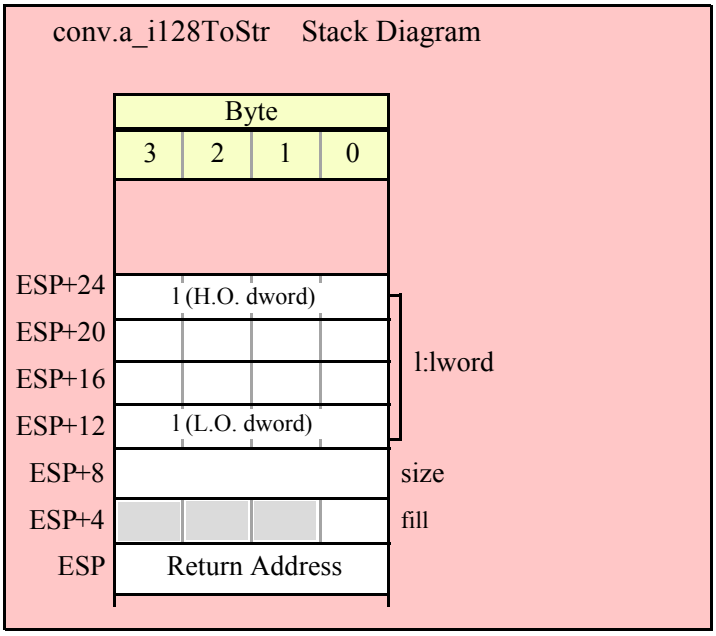
```
HLA high-level calling sequence examples:

// The following will push the value of "lwordVariable"
// and destStr onto the stack, and then call conv.i128ToStr:

conv.i128ToStr( lwordVariable, destStr );


// The following pushes the constant onto the stack and calls
// conv.i128ToStr:

conv.i128ToStr( <constant>, edx ); // EDX contains string pointer value.


HLA low-level calling sequence examples:

// Passing an lword variable:

  push( (type dword lwordVariable[12]));// H.O. dword first
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
push( destStr );
call conv.i128ToStr;


// Passing a constant:

pushd( <constant> >> 96 );// Push H.O. dword of constant first.
pushd( (<constant> >> 64)& $FFFF_FFFF );
pushd( (<constant> >> 32)& $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
push( edx );// EDX contains string pointer value.
call conv.i128ToStr;
```



```
procedure conv.a_i128ToStr( l:lword; width:int32; fill:char );
    @returns( "eax" );
```

This function converts a 128-bit signed integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push the value of "lwordVariable"
// onto the stack, and then call conv.a_i128ToStr:

conv.a_i128ToStr( lwordVariable );
mov( eax, destStr );


// The following pushes the constant onto the stack and calls
// conv.a_i128ToStr:

conv.a_i128ToStr( <constant> );
mov( eax, destStr );



HLA low-level calling sequence examples:

// Passing an lword variable:

  push( (type dword lwordVariable[12]));// H.O. dword first
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
call conv.a_i128ToStr;
mov( eax, destStr );


// Passing a constant:

pushd( <constant> >> 96 );// Push H.O. dword of constant first.
pushd( (<constant> >> 64)& $FFFF_FFFF );
pushd( (<constant> >> 32)& $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
call conv.a_i128ToStr;
mov( eax, destStr );
```



8.4.5 Signed Integer String to Numeric Conversions

The standard library string to integer conversion routines convert a sequence of digits, possibly prefaced by a minus sign, into the corresponding signed integer value. These routines begin by skipping over any leading delimiter characters (see the conv.getDelimiters and conv.setDelimiters functions for details), handling an optional minus sign, followed by any number of decimal digits and underscores (these routines ignore the underscores). Conversion stops at the end of the string or upon encountering a delimiter character.

These routines will raise a conversion error exception if they encounter a 7-bit ASCII character that is not a decimal digit, an underscore, or a delimiter character during the translation. These routines will raise an illegal character exception if they encounter a non-ASCII character (one with its H.O. bit set). These routines will raise a value out of range exception if the converted value will not fit in the destination data object.

There are two basic sets of string to numeric conversion routines: the conv.atoi* routines and the conv.strToi* routines. The atoi* routines process the characters pointed at by the ESI register. The strToi* routines process data in a string object, starting at an offset specified by a second parameter. For example, "conv.strToi8("12345", 3);" returns the value 45 because it begins processing the string at (zero-based) offset 3 in the string.

```
procedure conv.atoi8  ( buffer:var in esi ); @returns( "al" );
```

This function converts the sequence of characters starting at the memory address held in ESI to an 8-bit signed integer. It returns the value (in the range -128..+127) in AL. Note that this function actually returns the sign-extended value in EAX, so you may use EAX if it is more convenient to do so. Upon successful return, ESI is pointing at the delimiter character at the end of the sequence of digits.

```
HLA high-level calling sequence examples:

// The following will convert the decimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in AL:

conv.atoi8( [esi] );
mov( al, numericResult );
```

```
    // The following loads ESI with the address of
    // a sequence of decimal characters (held in an HLA
    // string) and converts them to an 8-bit number:

    conv.atoi8( sourceStr ); // Loads "sourceStr" into ESI
    mov( al, byteVariable );


    HLA low-level calling sequence examples:

      // Same as first example above - ESI already contains
      // the address of the first character to convert:

      call conv.atoi8;
    mov( al, numericResult );


    // Same as second example above

    static
      sourceStr :string := "12";
      .
      .
      .
  mov( sourceStr, esi );
   call conv.atoi8;
   mov( al, num12 );
```

procedure conv.atoi16 (buffer:var in esi); @returns("ax");

This function converts the sequence of characters starting at the memory address held in ESI to a 16-bit signed integer. It returns the value (in the range -32768..+32767) in AX. Note that this function actually returns the sign-extended value in EAX, so you may use EAX if it is more convenient to do so. Upon successful return, ESI is pointing at the delimiter character at the end of the sequence of digits.

```
    HLA high-level calling sequence examples:

    // The following will convert the decimal characters in memory
    // at the address specified by [esi] into a numeric value
    // and return that value in AX:

    conv.atoi16( [esi] );
    mov( ax, numericResult );


    // The following loads ESI with the address of
    // a sequence of decimal characters (held in an HLA
    // string) and converts them to a 16-bit number:

    conv.atoi16( sourceStr ); // Loads "sourceStr" into ESI
    mov( ax, wordVariable );


    HLA low-level calling sequence examples:

      // Same as first example above - ESI already contains
      // the address of the first character to convert:
```

```
    call conv.atoi16;
  mov( ax, numericResult );


  // Same as second example above

  static
    sourceStr :string := "12";
    .
    .
    .
mov( sourceStr, esi );
  call conv.atoi16;
  mov( ax, num12 );
```

procedure conv.atoi32 (buffer:var in esi); @returns("eax");

This function converts the sequence of characters starting at the memory address held in ESI to a 32-bit signed integer. It returns the value (in the range -2147483648..+2147483647) in EAX. ESI is pointing at the delimiter character at the end of the sequence of digits.

```
  HLA high-level calling sequence examples:

  // The following will convert the hexadecimal characters in memory
  // at the address specified by [esi] into a numeric value
  // and return that value in EAX:

  conv.atoi32( [esi] );
  mov( eax, numericResult );


  // The following loads ESI with the address of
  // a sequence of decimal characters (held in an HLA
  // string) and converts them to a 32-bit number:

  conv.atoi32( sourceStr ); // Loads "sourceStr" into ESI
  mov( eax, dwordVariable );


  HLA low-level calling sequence examples:

    // Same as first example above - ESI already contains
    // the address of the first character to convert:

    call conv.atoi32;
  mov( eax, numericResult );


  // Same as second example above

  static
    sourceStr :string := "12";
    .
    .
    .
mov( sourceStr, esi );
  call conv.atoi32;
  mov( eax, num12 );
```

procedure conv.atoi64 (buffer:var in esi); @returns("edx:eax");

This function converts the sequence of characters starting at the memory address held in ESI to a 64-bit signed integer. It returns the value (in the range -9223372036854775808..+9223372036854775807) in EDX:EAX (EDX contains the H.O. dword). ESI is pointing at the delimiter character at the end of the sequence of digits.

```
HLA high-level calling sequence examples:

// The following will convert the decimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in EDX:EAX:

conv.atoi64( [esi] );
mov( eax, (type dword hex64NumericResult[0]) );
mov( edx, (type dword hex64NumericResult[4]) );


// The following loads ESI with the address of
// a sequence of decimal characters (held in an HLA
// string) and converts them to a 64-bit number:

conv.atoi64( sourceStr ); // Loads "sourceStr" into ESI
mov( eax, (type dword qwordVariable[0]) );
mov( edx, (type dword qwordVariable[4]) );


HLA low-level calling sequence examples:

  // Same as first example above - ESI already contains
  // the address of the first character to convert:

  call conv.atoi64;
mov( eax, (type dword numericResult[0]) );
mov( edx, (type dword numericResult[4]) );


// Same as second example above

static
  sourceStr :string := "12";
  .
  .
  .
mov( sourceStr, esi );
 call conv.atoi64;
 mov( eax, (type dword qwordVariable[0]) );
 mov( edx, (type dword qwordVariable[4]) );
```

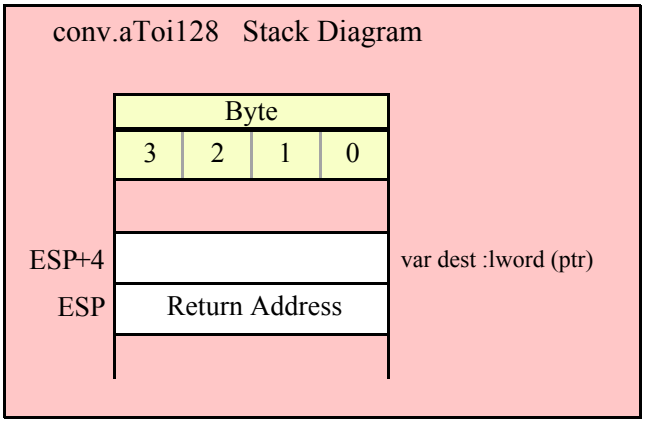
procedure conv.atoi128(buffer:var in esi; var l:lword);

This function converts the sequence of characters starting at the memory address held in ESI to a 128-bit signed integer. It returns the value (in the range -170141183460469231731687303715884105728..+170141183460469231731687303715884105727) in the l

parameter that is passed by reference to this function. ESI is pointing at the delimiter character at the end of the sequence of digits.

```
HLA high-level calling sequence examples:

// The following will convert the decimal characters in memory
// at the address specified by [esi] into a numeric value
// and stores that value in lwordDest (passed by reference):

conv.atoi128( [esi], lwordDest );


// The following loads ESI with the address of
// a sequence of decimal characters (held in an HLA
// string) and converts them to a 128-bit number that it
// stores in lwordDest:

conv.atoi128( sourceStr, lwordDest ); // Loads "sourceStr" into ESI



HLA low-level calling sequence examples:

  // Option 1: lwordDest is a static object declared in a
  // HLA STATIC, READONLY, or STORAGE section:
  // As with the first example above, assume ESI already
  // contains the address of the string to convert:

  pushd( &lwordDest );// Pass address of lwordDest as reference parm.
  call conv.atoi128;

  // Option 2: lwordDest is a simple automatic variable (no indexing)
  // declared in a VAR section (or as a parameter). Assume that
  // no 32-bit registers can be disturbed by this code.
  // As with the first example above, assume ESI already
  // contains the address of the string to convert:

  push( ebp );
  add( @offset( lwordDest ), (type dword [esp]));
  call conv.atoi128;

  // Option 3: lwordDest is a complex addressing mode and at least
  // one 32-bit register is available for use by this code.
  // As with the first example above, assume ESI already
  // contains the address of the string to convert:

  lea( eax, lwordDest );// Assume EAX is the available register
  push( eax );
  call conv.atoi128;


// Same as second high-level example above. Assumes that
// lwordDest is a static object.

static
  sourceStr :string := "12";
  .
  .
  .
mov( sourceStr, esi );
 pushd( &lwordDest );
```

```
call conv.atoi128;
```

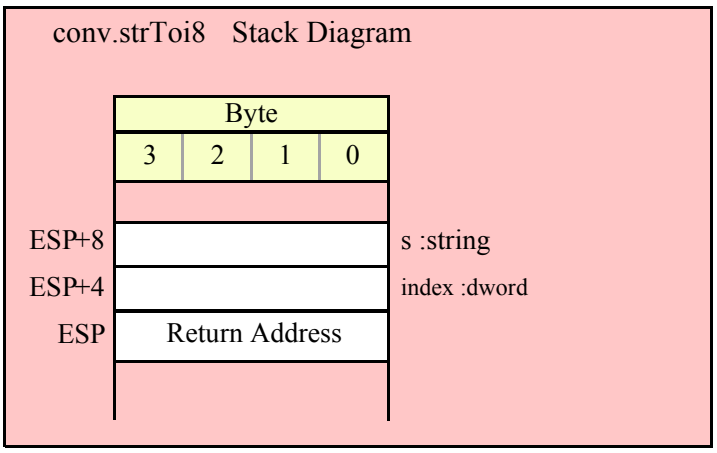


procedure conv.strToi8(s:string; index:dword)

This function converts the sequence of characters starting at zero-based offset index within the string parameter s to an 8-bit signed integer. It returns the value (in the range -128..+127) in AL. Note that this function actually returns the sign-extended value in EAX, so you may use EAX if it is more convenient to do so.

```
HLA high-level calling sequence examples:

// The following will convert the characters at the beginning
// of "decValueStr" to numeric form:

conv.strToi8( decValueStr, 0 );// Index=0 starts at beginning
mov( al, numericResult );


// The following demonstrates using a non-zero index:

conv.strToi8( "abc12", 3 ); // "12" begins at offset 3
mov( al, hex12 );


HLA low-level calling sequence examples:

  push( decValueStr );// Same as first example above
pushd( 0 );
call conv.strToi8;
mov( al, decNumericResult );


// Same as second example above

static
  str12 :string := "abc12";
  .
  .
  .
push( str12 );// Note that str12 points at "abc12".
```

```
pushd( 3 );// Index to "12" in "abc12".
call conv.strToi8;
mov( al, dec12 );
```

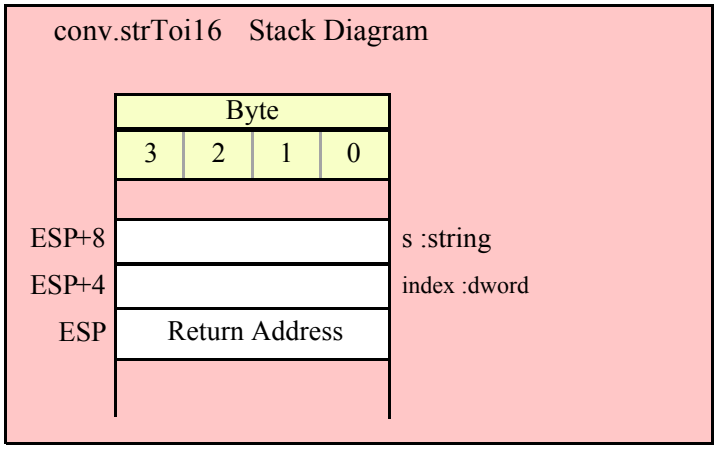


procedure conv.strToi16(s:string; index:dword)

This function converts the sequence of characters starting at zero-based offset index within the string parameter s to a 16-bit signed integer. It returns the value (in the range -32768..+32767) in AX. Note that this function actually returns the sign-extended value in EAX, so you may use EAX if it is more convenient to do so.

```
HLA high-level calling sequence examples:

// The following will convert the characters at the beginning
// of "decValueStr" to numeric form:

conv.strToi16( hexValueStr, 0 );// Index=0 starts at beginning
mov( ax, wordVar );


// The following demonstrates using a non-zero index:

conv.strToi16( "abc1234", 3 ); // "1234" begins at offset 3
mov( ax, wordVar );


HLA low-level calling sequence examples:

  push( decValueStr );// Same as first example above
pushd( 0 );
call conv.strToi16;
mov( ax, wordVar );


// Same as second example above

static
  str1200 :string := "abc1200";
  .
  .
```

```
  .
push( str1200 );// Note that str1200 points at "abc1200".
pushd( 3 );// Index to "1200" in "abc1200".
call conv.strToi16;
mov( ax, wordVar );
```

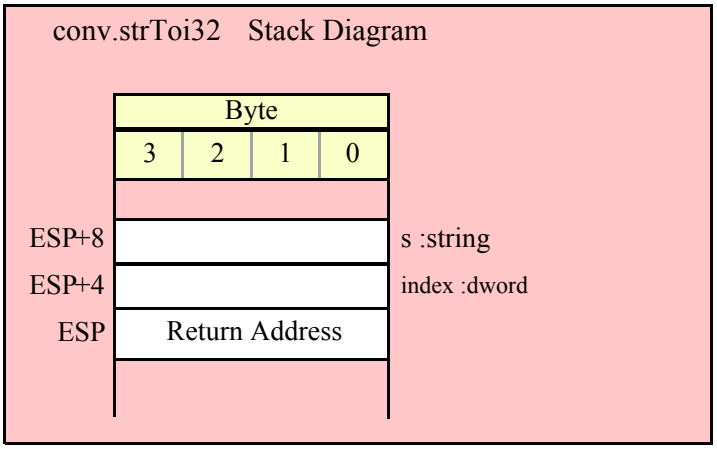


procedure conv.strToi32(s:string; index:dword)

This function converts the sequence of characters starting at zero-based offset index within the string parameter s to a 32-bit signed integer. It returns the value (in the range -2147483648..+2147483647) in EAX.

```
HLA high-level calling sequence examples:

// The following will convert the characters at the beginning
// of "decValueStr" to numeric form:

conv.strToi32( decValueStr, 0 );// Index=0 starts at beginning
mov( eax, dwordVar );


// The following demonstrates using a non-zero index:

conv.strToi32( "abc12_345", 3 ); // "12_345" begins at offset 3
mov( eax, dwordVar );


HLA low-level calling sequence examples:

  push( decValueStr );// Same as first example above
pushd( 0 );
call conv.strToi32;
mov( eax, dwordVar );


// Same as second example above

static
  str12345 :string := "abc-12_345";
  .
```

```
    .
    .
  push( str12345 );// Note that str12345 points at "abc-12_345".
  pushd( 3 ); // Index to "-12_345" in "abc-12_345".
  call conv.strToi32;
  mov( eax, dwordVar );// dwordVar now contains -12,345.
```

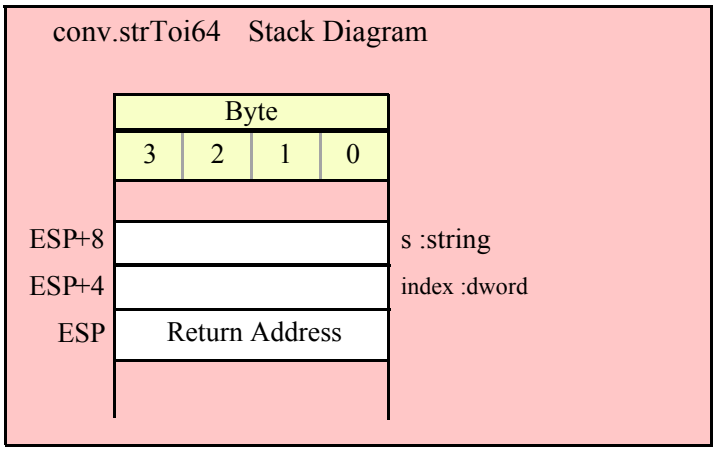


procedure conv.strToi64(s:string; index:dword)

This function converts the sequence of characters starting at zero-based offset index within the string parameter s to a 64-bit signed integer. It returns the value (in the range -9223372036854775808 .. +9223372036854775807) in EDX:EAX (EDX contains the H.O. dword).

```
  HLA high-level calling sequence examples:

  // The following will convert the characters at the beginning
  // of "decValueStr" to numeric form:

  conv.strToi64( decValueStr, 0 );// Index=0 starts at beginning
  mov( eax, (type dword qwordVar[0]) );
  mov( edx, (type dword qwordVar[4]) );


  // The following demonstrates using a non-zero index:

  conv.strToi64( "a-123", 1 ); // "-123" begins at offset 1
  mov( eax, (type dword qwordVar[0]) );
  mov( edx, (type dword qwordVar[4]) );



  HLA low-level calling sequence examples:

    push( decValueStr );// Same as first example above
  pushd( 0 );
  call conv.strToi64;
  mov( eax, (type dword qwordVar[0]) );
  mov( edx, (type dword qwordVar[4]) );
```

```
// Same as second example above

static
  strabc12 :string := "a-123";
  .
  .
  .
push( strabc12 );// Note that strabc12 points at "a-123".
pushd( 1 ); // Index to "-123" in "a-123".
call conv.strToi64;
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );
```

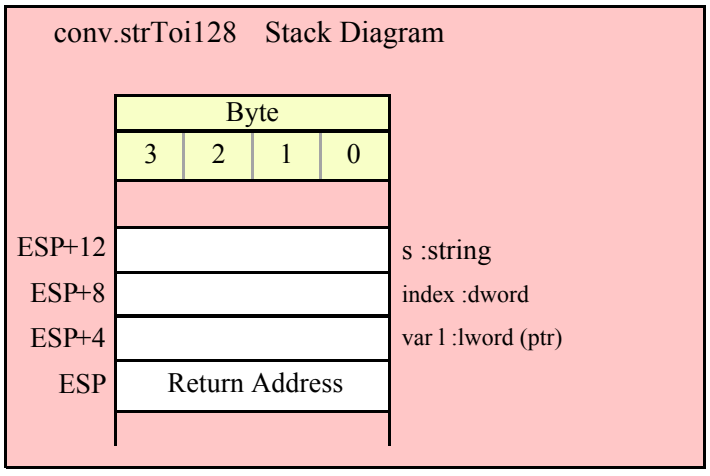


procedure conv.strToi128(s:string; index:dword; var dest:lword)

This function converts the sequence of characters starting at zero-based offset index within the string parameter s to a 128-bit signed integer. It returns the value (in the range -170141183460469231731687303715884105728.. +170141183460469231731687303715884105727) in the parameter l that you pass by reference to this function.

```
HLA high-level calling sequence examples:

// The following will convert the characters at the beginning
// of "decValueStr" (index=0) to numeric form and store the
// 128-bit result into the 1wordDest variable:

conv.strToi128( decValueStr, 0, 1wordDest );


// The following demonstrates using a non-zero index:

conv.strToi128( "abc1234567890123456789", 3, 1wordDest );


HLA low-level calling sequence examples:

  //  Option #1: lwordDest is a STATIC/READONLY/STORAGE
  // variable:

  push( decValueStr );// Same as first example above
```

```
    pushd( 0 );
    pushd( &lwordDest );
    call conv.strToi128;

      // Option #2: lwordDest is not a static object and
      // a 32-bit register is available for use:

      push( decValueStr );// Same as first example above
    pushd( 0 );
    lea( eax, lwordDest ); // Assuming EAX is available
    push( eax );
    call conv.strToi128;

      // Option #3: lwordDest is an automatic (var) object and
      // no 32-bit registers are available for use:

      push( decValueStr );// Same as first example above
    pushd( 0 );
    push( ebp );
    add( @offset( lwordDest ), (type dword [esp]) );
    call conv.strToi128;

      // Option #4: lwordDest is a complex addressing mode object and
      // no 32-bit registers are available for use:

      push( decValueStr );// Same as first example above
    pushd( 0 );
    sub( 4, esp );
    push( eax );
    lea( eax, lwordDest );
    mov( eax, [esp+4] );
    pop( eax );
    call conv.strToi128;
```



8.5 Unsigned Integer Conversions

The integer conversion function process signed integer values that are 8, 16, 32, 64, or 128 bits long. Functions in this category compute the output size (in print positions) of an integer, convert an integer to a sequence of characters, and convert a sequence of characters to an integer value.

8.5.1 Internal Routines

The following routines are used internally by the standard library unsigned integer code and you should not directly call them: conv._u8Size, conv._u16Size, and conv._u32Size.

8.5.2 Unsigned Integer Size Calculations

These routines return the size, in screen print positions, it would take to print the unsigned integer passed in the specified parameter. They return their value in the EAX register. Note that these routines include print positions required by underscores if you've enabled underscore output in values (see conv.setUnderscores and conv.getUnderscores for details).

It should go without saying that if you compute the size of an unsigned integer and then change the value of the internal underscores flag, the size you've computed may be invalid.

procedure conv.u8Size(b:byte in al); @returns("eax");

Computes the output size of an 8-bit unsigned integer (passed in AL) and returns this value in EAX. The return result will always be a value in the range 1..3. The internal underscores flag does not affect the result this function returns.

```
HLA high-level calling sequence examples:

conv.u8Size( byteVariable );
mov( eax, numSize );

conv.u8Size( <byte register> ); // al, ah, bl, bh, cl, ch, dl, dh
mov( eax, int8Size );

conv.u8Size( <constant> );      // Must fit into eight bits
mov( al, constantsSize );
```

Because conv.u8Size passes its input parameter in the AL register, any form of the high-level calling sequence except "conv.u8Size(al);" will automatically generate an instruction of the form "mov(<*operand*>,al);". Therefore, if possible, you should try to have the value whose size you wish to compute already sitting in the AL register and pass AL as the parameter to conv.u8Size.

```
HLA low-level calling sequence examples:

mov( byteVariable, al );
call conv.u8Size;
mov( eax, numSize );

mov( <byte register>, al ); // ah, bl, bh, cl, ch, dl, dh
call conv.u8Size;
mov( ax, wordVariable );

// Explicit Examples:

mov( bh, al );
call conv.u8Size;
mov( al, bhSize );

call conv.u8Size; // Assume value is already in AL
mov( al, alSize );


mov( 123, al );    // Example of computing the size of a constant
```

```
call conv.u8Size;
mov( eax, constsSize );
```

It might seem silly to compute the size of a constant as this last example is doing, as the constant's print width is known at compile time. Note, however, that this sequence could appear as part of a macro expansion and the literal constant "123" could actually be the result of expanding a macro parameter.

procedure conv.u16Size(w:word in ax)

Computes the output size of a 16-bit unsigned integer (passed in AX) and returns this value in EAX. The return result will always be a value in the range 1..5 if the internal underscores flag contains false, 1..6 if the underscores flag contains true.

```
HLA high-level calling sequence examples:

conv.u16Size( wordVariable );
mov( eax, numSize );

conv.u16Size( <word register> ); // ax, bx, cx, dx, bp, sp, si, di
mov( eax, int16Size );

conv.u16Size( <constant> );      // Must fit into 16 bits
mov( al, constantsSize );
```

Because conv.u16Size passes its input parameter in the AX register, any form of the high-level calling sequence except "conv.u16Size(ax);" will automatically generate an instruction of the form "mov(<*operand*>,ax);". Therefore, if possible, you should try to have the value whose size you wish to compute already sitting in the AX register and pass AX as the parameter to conv.u16Size.

```
HLA low-level calling sequence examples:

mov( wordVariable, ax );
call conv.u16Size;
mov( eax, numSize );

mov( <word register>, ax ); // bx, cx, dx, bp, sp, si, or di
call conv.u16Size;
mov( ax, wordVariable );

// Explicit Examples:

mov( bx, ax );
call conv.u16Size;
mov( al, bxSize );

call conv.u16Size; // Assume value is already in AX
mov( al, axSize );


mov( 12345, ax );    // Example of computing the size of a constant
call conv.u16Size;
mov( eax, constsSize );
```

See the comment at the end of conv.i8Size about passing constants to these functions.

procedure conv.u32Size(d:dword in eax)

Computes the output size of a 32-bit unsigned integer (passed in EAX) and returns this value in EAX. The return result will always be a value in the range 1..10 if the internal underscores flag contains false, 1..11 if the underscores flag contains true.

```
HLA high-level calling sequence examples:

conv.u32Size( wordVariable );
mov( eax, numSize );

conv.u32Size( <dword register> ); // eax, ebx, ecx, edx,
mov( eax, int16Size )l// ebp, esp, esi, or edi

conv.u32Size( <constant> );      // Must fit into 32 bits
mov( al, constantsSize );
```

Because conv.u32Size passes its input parameter in the EAX register, any form of the high-level calling sequence except "conv.u32Size(eax);" will automatically generate an instruction of the form "mov*(<operand>*,eax);". Therefore, if possible, you should try to have the value whose size you wish to compute already sitting in the EAX register and pass EAX as the parameter to conv.u32Size.

```
HLA low-level calling sequence examples:

mov( dwordVariable, eax );
call conv.u32Size;
mov( eax, numSize );

mov( <dword register>, eax ); // ebx, ecx, edx,
call conv.u32Size;// ebp, esp, esi, or edi
mov( ax, wordVariable );

// Explicit Examples:

mov( ebx, eax );
call conv.u32Size;
mov( al, bxSize );

call conv.u32Size; // Assume value is already in AX
mov( al, axSize );


mov( 1234567890, eax );    // Example of computing
call conv.u32Size;// the size of a constant.
mov( eax, constsSize );
```

See the comment at the end of conv.u8Size about passing constants to these functions.

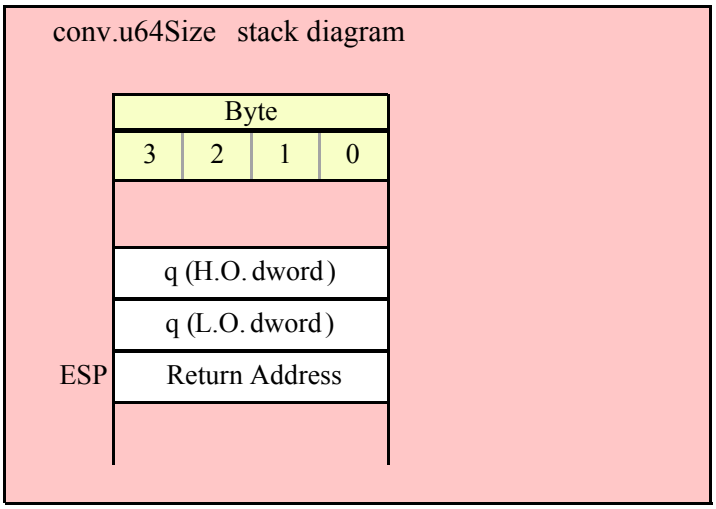
procedure conv.u64Size(q:qword)

Computes the output size of a 64-bit unsigned integer (passed in in q parameter) and returns this value in EAX. The return result will always be a value in the range 1..20 (e.g., "18446744073709551615") if the internal underscores flag contains false, 1..26 if the underscores flag contains true (e.g., "18_446_744_073_709_551_615").

```
HLA high-level calling sequence examples:

conv.u64Size( qwordVariable );
```

```
mov( eax, numSize );

conv.u64Size( <constant> );      // Must fit into 64 bits
mov( al, constantsSize );


HLA low-level calling sequence examples:

push( (type dword qwordVariable[4]));// Push H.O. dword first
push( (type dword qwordVariable[0]));// Push L.O. dword second
call conv.u64Size;
mov( eax, numSize );

// Compute the size of a 64-bit constant:

pushd( 12345 >> 32 );// Push H.O. dword first
pushd( 12345 & $FFFF_FFFF );// Push L.O. dword second
call conv.u64Size;
mov( eax, constsSize );
```

See the comment at the end of conv.u8Size about passing constants to these functions. If you make a habit of explicitly passing 64-bit constants to this function, you might consider writing a macro to push the 64-bit constant for you (see the chapter on "Passing Parameters to Standard Library Routines" for more details).



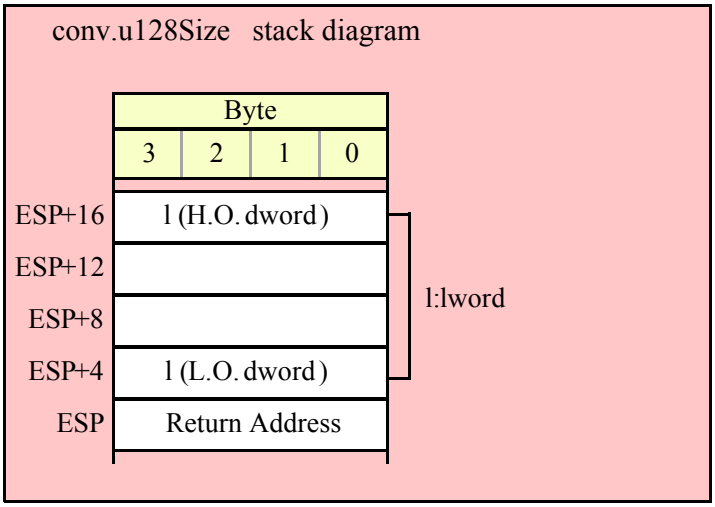
procedure conv.u128Size(l:lword)

Computes the output size of a 128-bit unsigned integer (passed in the l parameter) and returns this value in EAX. The return result will always be a value in the range 1..39 (e.g., "340282366920938463463374607431768211455") if the internal underscores flag contains false, 1..51 if the underscores flag contains true (e.g., "340_282_366_920_938_463_463_374_607_431_768_211_455").

```
HLA high-level calling sequence examples:

conv.u128Size( lwordVariable );
mov( eax, numSize );

conv.u128Size( <constant> );      // Must fit into 128 bits
mov( al, constantsSize );
```

```
HLA low-level calling sequence examples:

push( (type dword lwordVariable[12]));// Push H.O. dword first
push( (type dword lwordVariable[8]));
push( (type dword lwordVariable[4]));
push( (type dword lwordVariable[0]));// Push L.O. dword last
call conv.u64Size;
mov( eax, numSize );

// Compute the size of a 128-bit constant:

pushd( 12345 >> 96 );// Push H.O. dword first
pushd( (12345 >> 64) & $FFFF_FFFF );
pushd( (12345 >> 32 ) & $FFFF_FFFF );
pushd( 12345 & $FFFF_FFFF );// Push L.O. dword last
call conv.u128Size;
mov( eax, constsSize );
```

See the comment at the end of conv.u8Size about passing constants to these functions. If you make a habit of explicitly passing 128-bit constants to this function, you might consider writing a macro to push the 128-bit constant for you (see the chapter on "Passing Parameters to Standard Library Routines" for more details).



8.5.3 Unsigned Integer Numeric to Buffer Conversions

These routines convert the input parameter to a sequence of characters and store those characters starting at location [EDI]. They return EDI pointing at the first character beyond the converted string. Note that these functions do not zero terminate the string; if you want a zero-terminated string, then store a zero at the byte pointed at by EDI upon return from these functions.

If the internal underscores flag is set (see conv.getUnderscores and conv.setUnderscores for details), then these functions will insert an underscore between each group of three digits starting with the least significant digit.

`procedure conv.u8ToBuf( u8: uns8 in al )`

This function converts the 8-bit unsigned integer passed in AL to a sequence of 1..3 characters. The string this function produces is always in the range 0..255. Note that because this string always contains three or fewer digits, the internal underscores flag setting does not affect this function's output.

```
HLA high-level calling sequence examples:

// The following will load "byteVariable" into AL and
// the address of "charArrayVariable" into EDI and then
// call conv.u8ToBuf:

conv.u8ToBuf( byteVariable, charArrayVariable );

// The following call will copy BH into AL and
// EDX into EDI prior to calling conv.u8ToBuf:

conv.u8ToBuf( bh, [edx] );

// The following just calls conv.u8ToBuf as AL and EDI
// already hold the parameter values:

conv.u8ToBuf( al, [edi] );

// The following loads the constant in AL and calls
// conv.u8ToBuf:

conv.u8ToBuf( <constant>, [edi] ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the AL and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. byte of EAX (i.e., AL) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AL and [EDI].

```
HLA low-level calling sequence examples:

// Passing a byte variable and a buffer variable:

mov( byteVariable, al );
lea( edi, charArrayVariable );
call conv.u8ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( byteVariable, al );
mov( &charArrayVariable, edi );
call conv.u8ToBuf;

// Passing a pair of registers (that are not
// AL and EDI):

mov( bh, al );
mov( edx, edi );
call conv.u8ToBuf;

// Passing a constant:

mov( <constant>, al );
call conv.u8ToBuf; // Assume EDI already contains buffer address.
```

procedure conv.u16ToBuf(u16: uns16 in ax)

This function converts the 16-bit unsigned integer passed in AX to a sequence of 1..5 characters if the internal underscores flag is false, 1..6 characters if the underscores flag contains true. The string this function produces is always in the range 0..65535. If the internal underscores flag contains true and the value is greater than 999, then this function emits an underscore between the third and fourth digits (from the right) in the string.

```
HLA high-level calling sequence examples:

// The following will load "wordVariable" into AX and
// the address of "charArrayVariable" into EDI and then
// call conv.u16ToBuf:

conv.u16ToBuf( wordVariable, charArrayVariable );

// The following call will copy BX into AX and
// EDX into EDI prior to calling conv.u16ToBuf:

conv.u16ToBuf( bx, [edx] );

// The following just calls conv.u16ToBuf as AX and EDI
// already hold the parameter values:

conv.u16ToBuf( ax, [edi] );

// The following loads the constant in AX and calls
// conv.u16ToBuf:

conv.u16ToBuf( <constant>, [edi] ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the AX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. word of EAX (i.e., AX) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AX and [EDI].

```
HLA low-level calling sequence examples:

// Passing a word variable and a buffer variable:

mov( wordVariable, ax );
lea( edi, charArrayVariable );
call conv.u16ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( wordVariable, ax );
mov( &charArrayVariable, edi );
call conv.u16ToBuf;

// Passing a pair of registers (that are not
// AX and EDI):

mov( bx, ax );
mov( edx, edi );
call conv.u16ToBuf;

// Passing a constant:

mov( <constant>, ax );
```

```
call conv.u16ToBuf; // Assume EDI already contains buffer address.
```

procedure conv.u32ToBuf(u32: uns32 in eax)

This function converts the 32-bit unsigned integer passed in EAX to a sequence of 1..10 characters if the internal underscores flag is false, 1..11 characters if the underscores flag contains true. The string this function produces is always in the range 0..4294967295. If the internal underscores flag contains true and the value is greater than 999, then this function emits an underscore between the third and fourth digits (from the right) in the string.

```
HLA high-level calling sequence examples:

// The following will load "dwordVariable" into EAX and
// the address of "charArrayVariable" into EDI and then
// call conv.u32ToBuf:

conv.u32ToBuf( dwordVariable, charArrayVariable );

// The following call will copy EBX into EAX and
// EDX into EDI prior to calling conv.u32ToBuf:

conv.u32ToBuf( ebx, [edx] );

// The following just calls conv.u32ToBuf as EAX and EDI
// already hold the parameter values:

conv.u32ToBuf( eax, [edi] );

// The following loads the constant in EAX and calls
// conv.u32ToBuf:

conv.u32ToBuf( <constant>, [edi] ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EAX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite EAX before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not EAX and [EDI].

```
HLA low-level calling sequence examples:

// Passing a dword variable and a buffer variable:

mov( dwordVariable, eax );
lea( edi, charArrayVariable );
call conv.u32ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( dwordVariable, eax );
mov( &charArrayVariable, edi );
call conv.u32ToBuf;

// Passing a pair of registers (that are not
// EAX and EDI):

mov( ebx, eax );
mov( edx, edi );
```

```
call conv.u32ToBuf;

// Passing a constant:

mov( <constant>, eax );
call conv.u32ToBuf; // Assume EDI already contains buffer address.
```

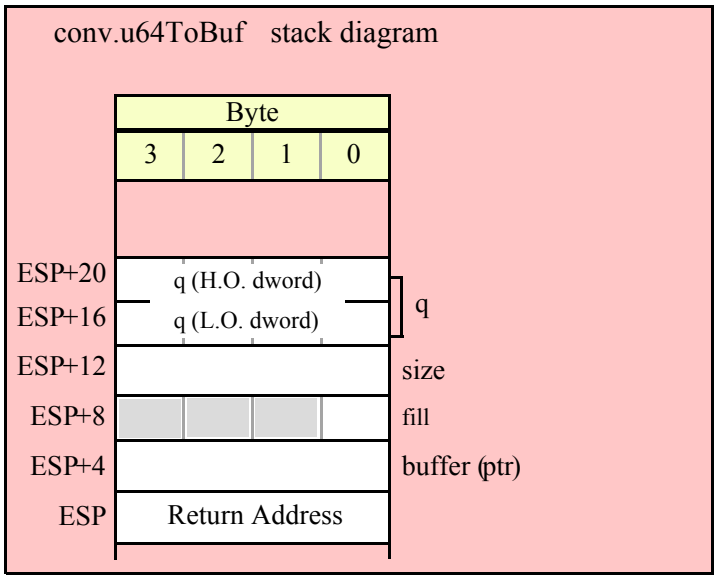
procedure conv.u64ToBuf(q:qword)

This function converts the 64-bit unsigned integer passed in q to a sequence of 1..20 characters if the internal underscores flag is false, 1..26 characters if the underscores flag contains true. The string this function produces is always in the range 0 .. 18446744073709551615. If the internal underscores flag contains true and the value is greater than 999, then this function emits an underscore between each group of three digits starting with the least significant digit.

```
HLA high-level calling sequence examples:

// The following will push the value of "qwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.u64ToBuf:

conv.u64ToBuf( qwordVariable, charArrayVariable );


// The following pushes the constant onto the stack and calls
// conv.u64ToBuf:

conv.u64ToBuf( <constant>, [edi] ); // <constant> must fit in 64 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

```
HLA low-level calling sequence examples:

// Passing a qword variable and a buffer variable:

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
lea( edi, charArrayVariable );
call conv.u64ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
mov( &charArrayVariable, edi );
call conv.u64ToBuf;

// Passing a constant:

pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
call conv.u64ToBuf; // Assume EDI already contains buffer address.
```



procedure conv.u128ToBuf(l:lword)

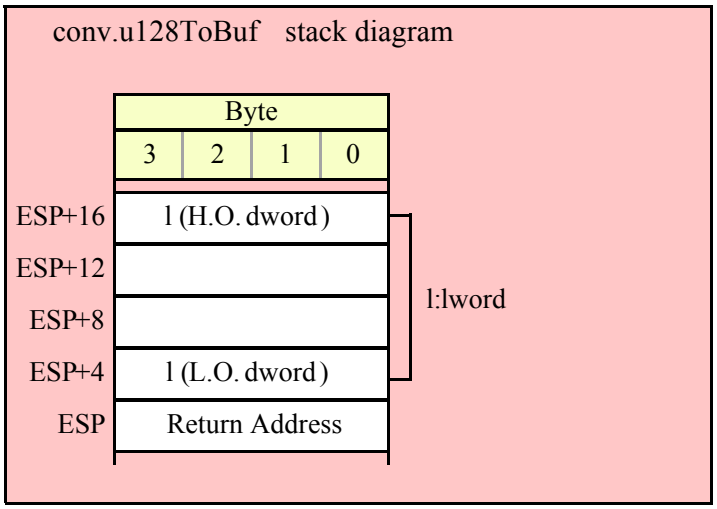
This function converts the 128-bit unsigned integer passed in l to a sequence of 1..39 characters if the internal underscores flag is false, 1..52 characters if the underscores flag contains true. The string this function produces is always in the range 0 .. 340282366920938463463374607431768211455. If the internal underscores flag contains true and the value is greater than 999, then this function emits an underscore between each group of three digits starting with the least significant digit.

```
HLA high-level calling sequence examples:

// The following will push the value of "lwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.u128ToBuf:

conv.u128ToBuf( lwordVariable, charArrayVariable );


// The following pushes the constant onto the stack and calls
// conv.u128ToBuf:

conv.u128ToBuf( <constant>, [edi] ); // <constant> must fit in 128 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

```
HLA low-level calling sequence examples:

// Passing an lword variable and a buffer variable:

  push( (type dword lwordVariable[12]));// H.O. dword first
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
lea( edi, charArrayVariable );
```

```
call conv.u128ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

  push( (type dword lwordVariable[12]));// H.O. dword first
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
mov( &charArrayVariable, edi );
call conv.u128ToBuf;

// Passing a constant:

pushd( <constant> >> 96 );// Push H.O. dword of constant first.
pushd( (<constant> >> 64)& $FFFF_FFFF );
pushd( (<constant> >> 32)& $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
call conv.u128ToBuf; // Assume EDI already contains buffer address.
```



8.5.4 Unsigned Integer Numeric to String Conversions

These routines convert an unsigned integer value (8, 16, 32, 64, or 128 bits) to a string. The standard ("unadorned") functions store the string data into a string object that you pass as a parameter to the function. That string object must be preallocated and large enough to receive the string result (else a string overflow occurs). The "adorned" functions, whose names begin with "a_" automatically allocate storage on the heap, store the converted string into that heap object, and then return a pointer to the newly allocated string in the EAX register (it is the caller's responsibility to free the storage when it is no longer needed).

These functions let you specify a minimum field width and a fill character. If the number would require fewer than width print positions, the routines copy the fill character to the remaining positions in the destination string. If width is positive, the number is right justified in the string. If width is negative, the number is left justified in the string. If the string representation of the value requires more than width print positions, then these functions ignore the width and fill paramenters and use however many positions are necessary to properly display the value.

xxxToStr (value, width, fill, buffer);

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

V	A	L	U	E	f	f	f

Here are the maximum number of print positions these routines will produce for each data type before considering the minimum field width:

```
Underscores flag is false:
  8 bits:3 (0..255)
  16 bits:5 (0..65535)
  32 bits:10 (0..4294967295)
  64 bits:20 (0..18446744073709551615)
  128 bits:39 (0 .. 340282366920938463463374607431768211455)

Underscores flag is true:
  8 bits:3 (0..255)
  16 bits:6 (0..65_535)
  32 bits:13 (0..4_294_967_295)
  64 bits:26 (0..18_446_744_073_709_551_615)
  128 bits:51 (0..340_282_366_920_938_463_463_374_607_431_768_211_455)
```

procedure conv.u8ToStr (b:uns8; width:int32; fill:char; buffer:string);

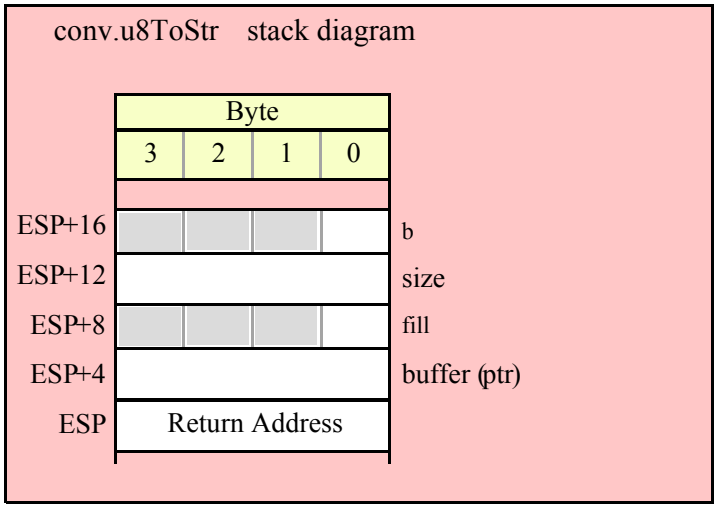
This function converts an 8-bit unsigned integer to the decimal string representation of that integer and stores the string in the preallocated string object specified by the dest paramenter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. Note that the internal underscores flag will not affect the output because 8-bit integers are always three digits or smaller.

```
HLA high-level calling sequence examples:

// The following will push "byteVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.u8ToStr:

conv.u8ToStr( byteVariable, destStr );

// The following call will BH's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.u8ToStr:

conv.u8ToStr( bh, edx );

// The following pushes the constant and destStr and calls
// conv.u8ToStr:

conv.bToBuf( <constant>, destStr ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns

out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

```
HLA low-level calling sequence examples:

// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
push( destStr );
call conv.u8ToStr;

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
push( destStr );
call conv.u8ToStr;

// Passing a byte variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
push( destStr );
call conv.u8ToStr;


// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BL
push( edx );
call conv.u8ToStr;

// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
mov( bh, [esp] );
push( edx );
call conv.u8ToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.u8ToStr;
```

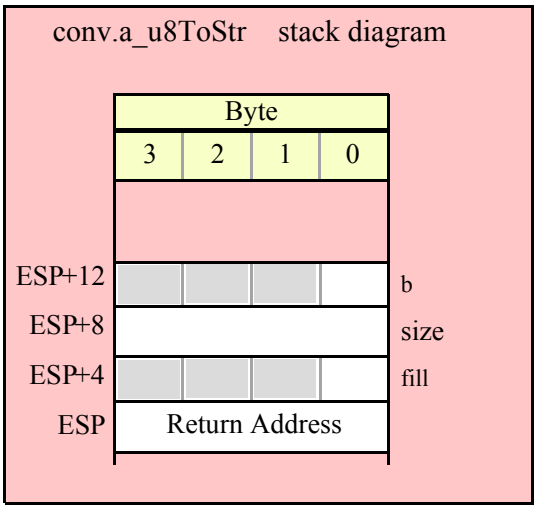



```
procedure conv.a_u8ToStr ( b:uns8;  width:int32; fill:char );
    @returns( "eax" );
```

This function converts an 8-bit unsigned integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. Note that the internal underscores flag will not affect the output because 8-bit integers are always three digits or smaller.

```
HLA high-level calling sequence examples:

// The following will push "byteVariable" and then call conv.a_u8ToStr:

conv.a_u8ToStr( byteVariable );
mov( eax, byteStr );

// The following call will BH's value onto the stack
// before calling conv.a_u8ToStr:

conv.a_u8ToStr( bh );
mov( eax, byteStr );

// The following pushes the constant and calls
// conv.a_u8ToStr:

conv.a_u8ToStr( <constant> ); // <constant> must fit in 8 bits
mov( eax, byteStr );


HLA low-level calling sequence examples:

// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
```

```
call conv.a_u8ToStr;
mov( eax, destStr );

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
call conv.a_u8ToStr;
mov( eax, destStr );

// Passing a byte variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
call conv.a_u8ToStr;
mov( eax, destStr );

// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BL
call conv.a_u8ToStr;
mov( eax, byteStr );

// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
mov( bh, [esp] );
call conv.a_u8ToStr;
mov( eax, byteStr );

// Passing a constant:

pushd( <constant> );
call conv.a_u8ToStr;
mov( eax, byteStr );
```



procedure conv.u16ToStr(w:uns16; width:int32; fill:char; buffer:string);

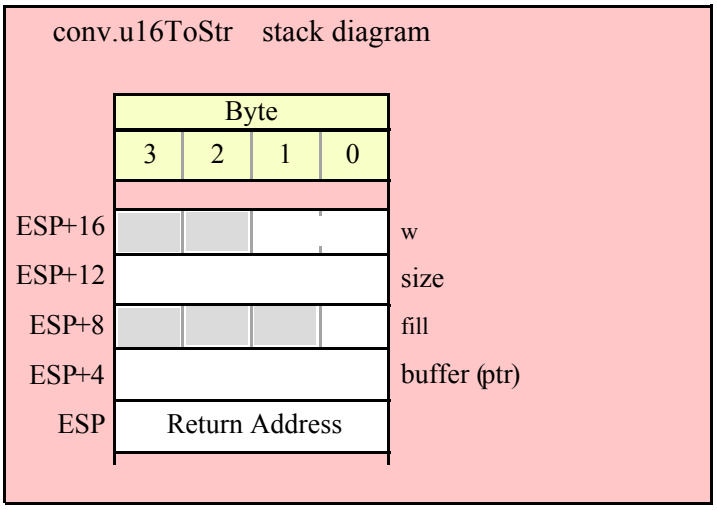
This function converts a 16-bit unsigned integer to its decimal string representation and stores the string in the preallocated string object specified by the dest paramenter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push "wordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.u16ToStr:

conv.u16ToStr( wordVariable, destStr );

// The following call will BX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.u16ToStr:

conv.u16ToStr( bx, edx );

// The following pushes the constant and destStr and calls
// conv.u16ToStr:

conv.u16ToStr( <constant>, destStr ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

```
HLA low-level calling sequence examples:

// Passing a word variable and a buffer variable, option 1
```

```
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
push( destStr );
call conv.u16ToStr;

// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

push( (type dword wordVariable));
push( destStr );
call conv.u16ToStr;

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:

pushw( 0 );
push( wordVariable );
push( destStr );
call conv.u16ToStr;


// Passing a pair of registers:
// BX = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BX
push( edx );
call conv.u16ToStr;


// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.u16ToStr;
```

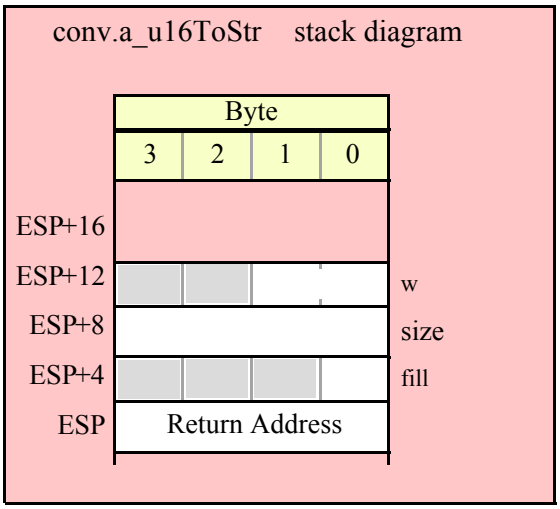


procedure conv.a_u16ToStr(w:uns16; width:int32; fill:char);
@returns("eax");

This function converts a 16-bit unsigned integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push "wordVariable" and call conv.a_u16ToStr:

conv.a_u16ToStr( wordVariable );
mov( eax, destStr );

// The following call will BX's value onto the stack
// before calling conv.a_u16ToStr:

conv.a_u16ToStr( bx );

// The following pushes the constant and calls
// conv.a_u16ToStr:

conv.a_u16ToStr( <const>, destStr ); // <const> must fit in 16 bits



HLA low-level calling sequence examples:

// Passing a word variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( wordVariable, eax );
push( eax );
call conv.a_u16ToStr;
mov( eax, destStr );

// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

push( (type dword wordVariable));
call conv.a_u16ToStr;
mov( eax, destStr );

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:

pushw( 0 );
push( wordVariable );
call conv.a_u16ToStr;
mov( eax, destStr );


// Passing a pair of registers:
// BX = value to print.
```

```
push( ebx );      // Pushes BX
call conv.a_u16ToStr;
mov( eax, wordStr );


// Passing a constant:

pushd( <constant> );
call conv.a_u16ToStr;
mov( eax, destStr );
```



procedure conv.u32ToStr(d:uns32; width:int32; fill:char; buffer:string);

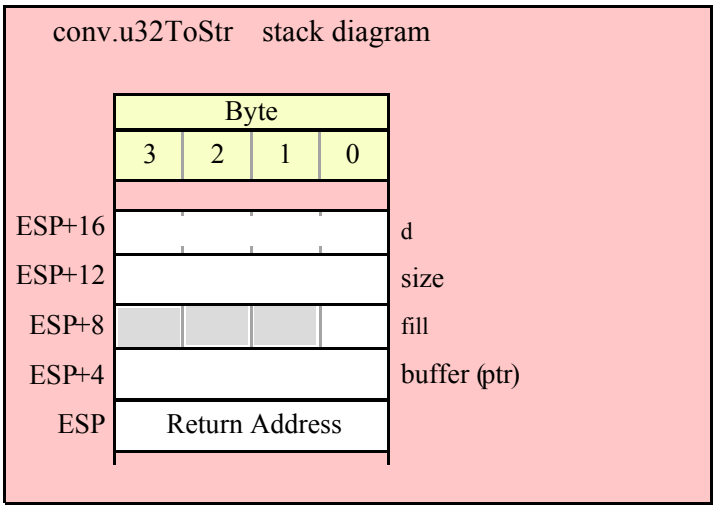
This function converts a 32-bit unsigned integer to its decimal string representation and stores the string in the preallocated string object specified by the buffer paramenter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push "dwordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.u32ToStr:

conv.u32ToStr( dwordVariable, destStr );

// The following call will push EBX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.u32ToStr:

conv.u32ToStr( ebx, edx );

// The following pushes the constant and destStr and calls
// conv.u32ToStr:
```

```
conv.u32ToStr( <constant>, destStr ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data. In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string data.

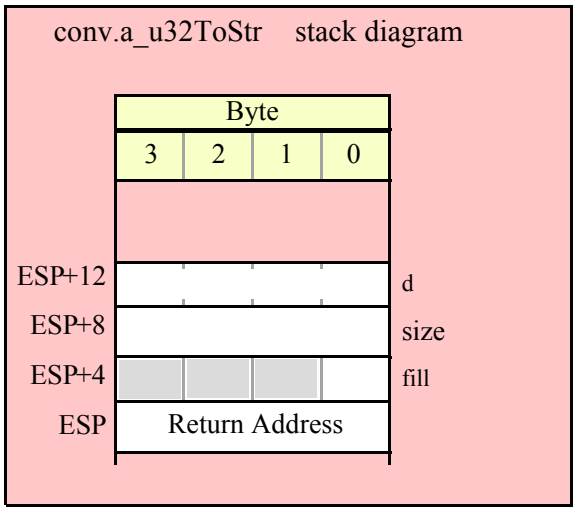
```
HLA low-level calling sequence examples:


// Passing a dword variable and a buffer variable:

push( dwordVariable );
push( destStr );
call conv.u32ToStr;


// Passing a pair of registers:
// EBX = value to print, EDX = pointer to string object.

push( ebx );
push( edx );
call conv.u32ToStr;


// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.u32ToStr;
```



```
procedure conv.a_u32ToStr( d:uns32; width:int32; fill:char );
    @returns( "eax" );
```

This function converts a 32-bit unsigned integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push "dwordVariable" and then call conv.a_u32ToStr:

conv.a_u32ToStr( dwordVariable );
mov( eax, destStr );

// The following call will push EBX's value onto the stack
// before calling conv.a_u32ToStr:

conv.a_u32ToStr( ebx );

// The following pushes the constant and calls
// conv.a_u32ToStr:

conv.a_u32ToStr( <constant>, destStr ); // <constant> must fit in 32 bits



HLA low-level calling sequence examples:

// Passing a dword variable:

push( dwordVariable );
call conv.a_u32ToStr;
mov( eax, destStr );


// Passing a register:
// EBX = value to print.

push( ebx );
call conv.a_u32ToStr;
mov( eax, dwordStr );


// Passing a constant:

pushd( <constant> );
call conv.a_u32ToStr;
mov( eax, destStr );
```

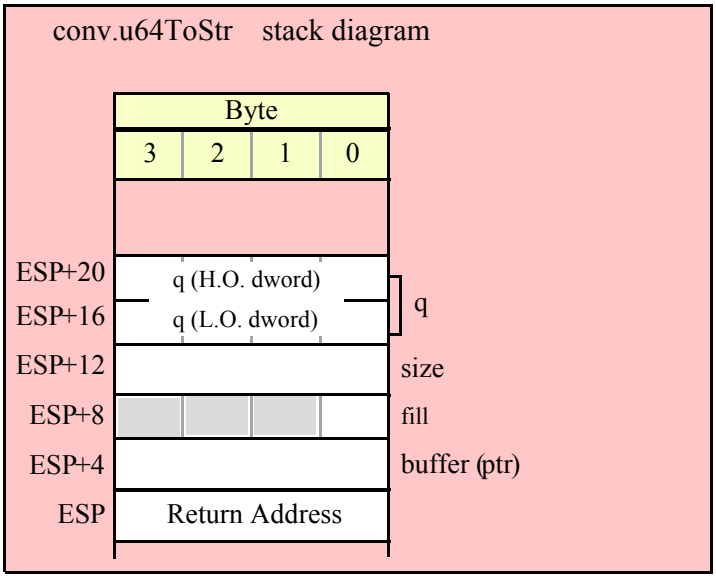


procedure conv.u64ToStr(q:qword; width:int32; fill:char; buffer:string);

This function converts a 64-bit unsigned integer to its decimal string representation and stores the string in the preallocated string object specified by the buffer paramenter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push the value of "qwordVariable"
// and the value of the destStr string variable
// onto the stack and then call conv.u64ToStr:

conv.u64ToStr( qwordVariable, destStr );


// The following pushes the constant onto the stack along with
// the value held in the destStr variable and calls
// conv.u64ToStr:

conv.u64ToStr( <constant>, destStr ); // <constant> must fit in 64 bits


HLA low-level calling sequence examples:

// Passing a qword variable and a buffer variable:

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
push( destStr );
call conv.u64ToStr;
```

```
// Passing a constant:

pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
push( destStr );
call conv.u64ToStr;
```

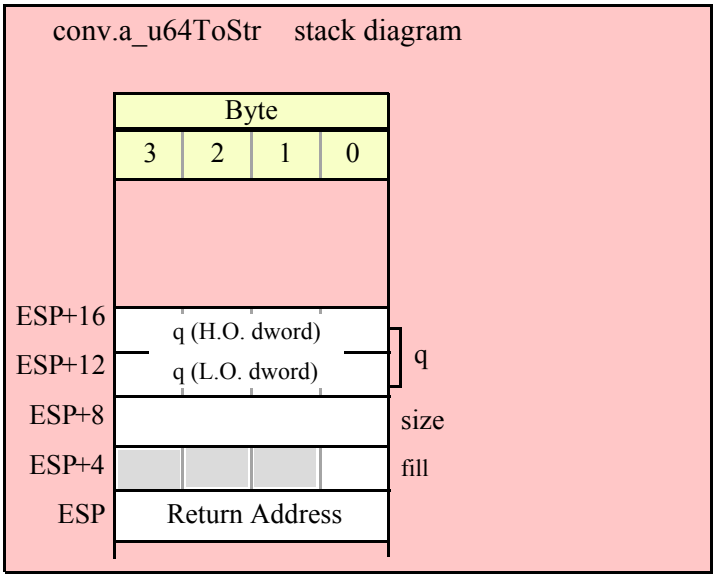


procedure conv.a_u64ToStr(q:qword; width:int32; fill:char);
@returns("eax");

This function converts a 64-bit unsigned integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push the value of "qwordVariable"
// onto the stack and then call conv.a_u64ToStr:

conv.a_u64ToStr( qwordVariable );
mov( eax, destStr );


// The following pushes the constant onto the stack and calls
// conv.a_u64ToStr:

conv.a_u64ToStr( <constant> ); // <constant> must fit in 64 bits
mov( eax, destStr );


HLA low-level calling sequence examples:
```

```
// Passing a qword variable:

  push( (type dword qwordVariable[4]));// H.O. dword first
  push( (type dword qwordVariable[0]));// L.O. dword last
call conv.a_u64ToStr;
mov( eax, destStr );


// Passing a constant:

pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
call conv.a_u64ToStr;
mov( eax, destStr );
```

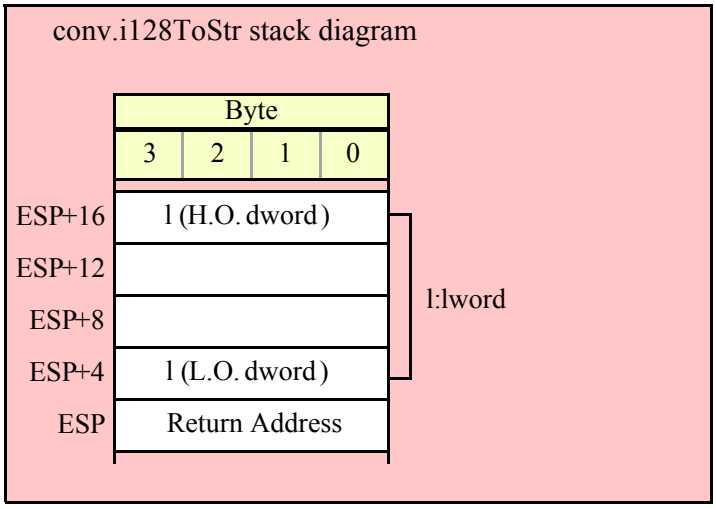


procedure conv.u128ToStr(l:lword; width:int32; fill:char; buffer:string);

This function converts a 128-bit unsigned integer to its decimal string representation and stores the string in the preallocated string object specified by the buffer paramenter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push the value of "lwordVariable"
// and destStr onto the stack, and then call conv.u128ToStr:

conv.u128ToStr( lwordVariable, destStr );


// The following pushes the constant onto the stack and calls
// conv.u128ToStr:
```

```
conv.u128ToStr( <constant>, edx ); // EDX contains string pointer value.


HLA low-level calling sequence examples:

// Passing an lword variable:

  push( (type dword lwordVariable[12]));// H.O. dword first
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
push( destStr );
call conv.u128ToStr;


// Passing a constant:

pushd( <constant> >> 96 );// Push H.O. dword of constant first.
pushd( (<constant> >> 64)& $FFFF_FFFF );
pushd( (<constant> >> 32)& $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
push( edx );// EDX contains string pointer value.
call conv.u128ToStr;
```

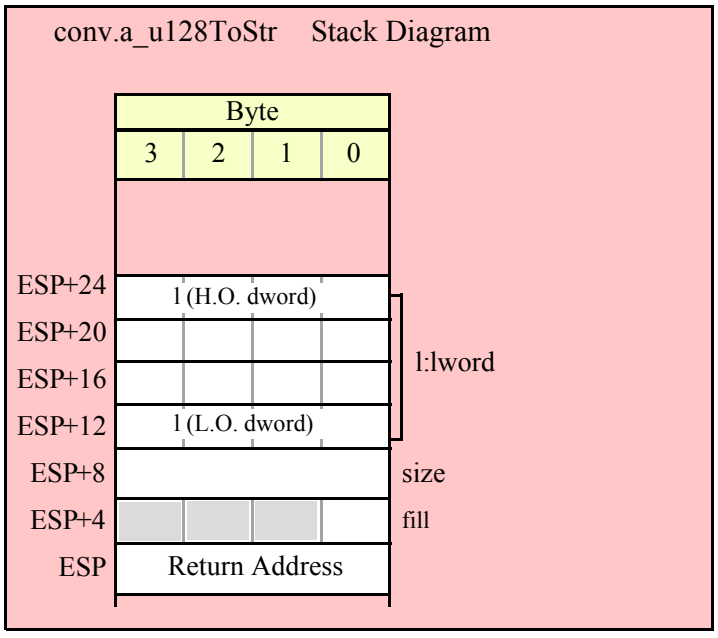


procedure conv.a_u128ToStr(l:lword; width:int32; fill:char);
@returns("eax");

This function converts a 128-bit signed integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least signficant digit (see conv.getUnderscores and conv.setUnderscores for more details).

```
HLA high-level calling sequence examples:

// The following will push the value of "lwordVariable"
// onto the stack, and then call conv.a_u128ToStr:
```

```
conv.a_u128ToStr( lwordVariable );
mov( eax, destStr );


// The following pushes the constant onto the stack and calls
// conv.a_u128ToStr:

conv.a_u128ToStr( <constant> );
mov( eax, destStr );


HLA low-level calling sequence examples:

// Passing an lword variable:

  push( (type dword lwordVariable[12]));// H.O. dword first
  push( (type dword lwordVariable[8]));
  push( (type dword lwordVariable[4]));
  push( (type dword lwordVariable[0]));// L.O. dword last
call conv.a_u128ToStr;
mov( eax, destStr );


// Passing a constant:

pushd( <constant> >> 96 );// Push H.O. dword of constant first.
pushd( (<constant> >> 64)& $FFFF_FFFF );
pushd( (<constant> >> 32)& $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
call conv.a_u128ToStr;
mov( eax, destStr );
```



8.5.5 Unsigned Integer String to Numeric Conversions

The standard library string to integer conversion routines convert a sequence of digits into the corresponding unsigned integer value. These routines begin by skipping over any leading delimiter characters (see the conv.getDelimiters and conv.setDelimiters functions for details) followed by any number of decimal digits and underscores (these routines ignore the underscores). Conversion stops at the end of the string or upon encountering a delimiter character.

These routines will raise a conversion error exception if they encounter a 7-bit ASCII character that is not a decimal digit, an underscore, or a delimiter character during the translation. These routines will raise an illegal character exception if they encounter a non-ASCII character (one with its H.O. bit set). These routines will raise a value out of range exception if the converted value will not fit in the destination data object.

There are two basic sets of string to numeric conversion routines: the conv.atou* routines and the conv.strTou* routines. The atou* routines process the characters pointed at by the ESI register. The strTou* routines process data in a string object, starting at an offset specified by a second parameter. For example, "conv.strTou8("12345", 3);" returns the value 45 because it begins processing the string at (zero-based) offset 3 in the string.

```
procedure conv.atou8  ( buffer: var in esi ); @returns( "ax" );
```

This function converts the sequence of characters starting at the memory address held in ESI to an 8-bit unsigned integer. It returns the value (in the range 0..+255) in AL. Note that this function actually returns the zero-extended value in EAX, so you may use EAX if it is more convenient to do so. Upon successful return, ESI is pointing at the delimiter character at the end of the sequence of digits.

```
HLA high-level calling sequence examples:

// The following will convert the decimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in AL:

conv.atou8( [esi] );
mov( al, numericResult );


// The following loads ESI with the address of
// a sequence of decimal characters (held in an HLA
// string) and converts them to an 8-bit number:

conv.atou8( sourceStr ); // Loads "sourceStr" into ESI
mov( al, byteVariable );



HLA low-level calling sequence examples:

  // Same as first example above - ESI already contains
  // the address of the first character to convert:

  call conv.atou8;
mov( al, numericResult );


// Same as second example above

static
  sourceStr :string := "12";
  .
  .
  .
mov( sourceStr, esi );
 call conv.atou8;
 mov( al, num12 );
```

procedure conv.atou16 (buffer: var in esi); @returns("ax");

This function converts the sequence of characters starting at the memory address held in ESI to a 16-bit unsigned integer. It returns the value (in the range 0..65535) in AX. Note that this function actually returns the zero-extended value in EAX, so you may use EAX if it is more convenient to do so. Upon successful return, ESI is pointing at the delimiter character at the end of the sequence of digits.

```
HLA high-level calling sequence examples:

// The following will convert the decimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in AX:

conv.atou16( [esi] );
mov( ax, numericResult );


// The following loads ESI with the address of
// a sequence of decimal characters (held in an HLA
// string) and converts them to a 16-bit number:

conv.atou16( sourceStr ); // Loads "sourceStr" into ESI
mov( ax, wordVariable );



HLA low-level calling sequence examples:

  // Same as first example above - ESI already contains
  // the address of the first character to convert:

  call conv.atou16;
mov( ax, numericResult );


// Same as second example above

static
  sourceStr :string := "12";
  .
  .
  .
mov( sourceStr, esi );
 call conv.atou16;
 mov( ax, num12 );
```

procedure conv.atou32 (buffer: var in esi); @returns("eax");

This function converts the sequence of characters starting at the memory address held in ESI to a 32-bit unsigned integer. It returns the value (in the range 0..4294967295) in EAX. ESI is pointing at the delimiter character at the end of the sequence of digits.

```
HLA high-level calling sequence examples:

// The following will convert the hexadecimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in EAX:
```

```
    conv.atou32( [esi] );
    mov( eax, numericResult );


    // The following loads ESI with the address of
    // a sequence of decimal characters (held in an HLA
    // string) and converts them to a 32-bit number:

    conv.atou32( sourceStr ); // Loads "sourceStr" into ESI
    mov( eax, dwordVariable );



    HLA low-level calling sequence examples:

      // Same as first example above - ESI already contains
      // the address of the first character to convert:

      call conv.atou32;
    mov( eax, numericResult );


    // Same as second example above

    static
      sourceStr :string := "12";
      .
      .
      .
  mov( sourceStr, esi );
    call conv.atou32;
    mov( eax, num12 );
```

procedure conv.atou64 (buffer: var in esi); @returns("edx:eax");

This function converts the sequence of characters starting at the memory address held in ESI to a 64-bit unsigned integer. It returns the value (in the range 0..18446744073709551615) in EDX:EAX (EDX contains the H.O. dword). ESI is pointing at the delimiter character at the end of the sequence of digits.

```
    HLA high-level calling sequence examples:

    // The following will convert the decimal characters in memory
    // at the address specified by [esi] into a numeric value
    // and return that value in EDX:EAX:

    conv.atou64( [esi] );
    mov( eax, (type dword hex64NumericResult[0]) );
    mov( edx, (type dword hex64NumericResult[4]) );


    // The following loads ESI with the address of
    // a sequence of decimal characters (held in an HLA
    // string) and converts them to a 64-bit number:

    conv.atou64( sourceStr ); // Loads "sourceStr" into ESI
    mov( eax, (type dword qwordVariable[0]) );
    mov( edx, (type dword qwordVariable[4]) );
```

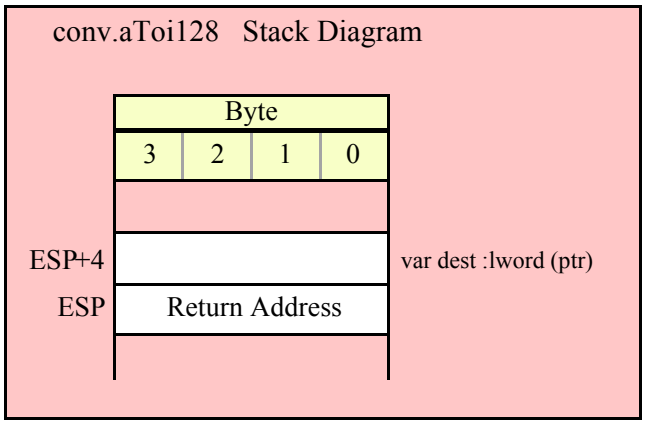
```
  HLA low-level calling sequence examples:

    // Same as first example above - ESI already contains
    // the address of the first character to convert:

    call conv.atou64;
  mov( eax, (type dword numericResult[0]) );
  mov( edx, (type dword numericResult[4]) );


  // Same as second example above

  static
    sourceStr :string := "12";
    .
    .
    .
mov( sourceStr, esi );
  call conv.atou64;
  mov( eax, (type dword qwordVariable[0]) );
  mov( edx, (type dword qwordVariable[4]) );
```

procedure conv.atou128(buffer: var in esi; var l:lword);

This function converts the sequence of characters starting at the memory address held in ESI to a 128-bit signed integer. It returns the value (in the range 0..340282366920938463463374607431768211455) in the l parameter that is passed by reference to this function. ESI is pointing at the delimiter character at the end of the sequence of digits.

```
  HLA high-level calling sequence examples:

  // The following will convert the decimal characters in memory
  // at the address specified by [esi] into a numeric value
  // and stores that value in lwordDest (passed by reference):

  conv.atou128( [esi], lwordDest );


  // The following loads ESI with the address of
  // a sequence of decimal characters (held in an HLA
  // string) and converts them to a 128-bit number that it
  // stores in lwordDest:

  conv.atou128( sourceStr, lwordDest ); // Loads "sourceStr" into ESI


  HLA low-level calling sequence examples:

    // Option 1: lwordDest is a static object declared in a
    // HLA STATIC, READONLY, or STORAGE section:
    // As with the first example above, assume ESI already
    // contains the address of the string to convert:

    pushd( &lwordDest );// Pass address of lwordDest as reference parm.
    call conv.atou128;

    // Option 2: lwordDest is a simple automatic variable (no indexing)
```

```
    // declared in a VAR section (or as a parameter). Assume that
    // no 32-bit registers can be disturbed by this code.
    // As with the first example above, assume ESI already
    // contains the address of the string to convert:

    push( ebp );
    add( @offset( lwordDest ), (type dword [esp]));
    call conv.atou128;

    // Option 3: lwordDest is a complex addressing mode and at least
    // one 32-bit register is available for use by this code.
    // As with the first example above, assume ESI already
    // contains the address of the string to convert:

    lea( eax, lwordDest );// Assume EAX is the available register
    push( eax );
    call conv.atou128;


  // Same as second high-level example above. Assumes that
  // lwordDest is a static object.

  static
    sourceStr :string := "12";
    .
    .
    .
mov( sourceStr, esi );
 pushd( &lwordDest );
 call conv.atou128;
```

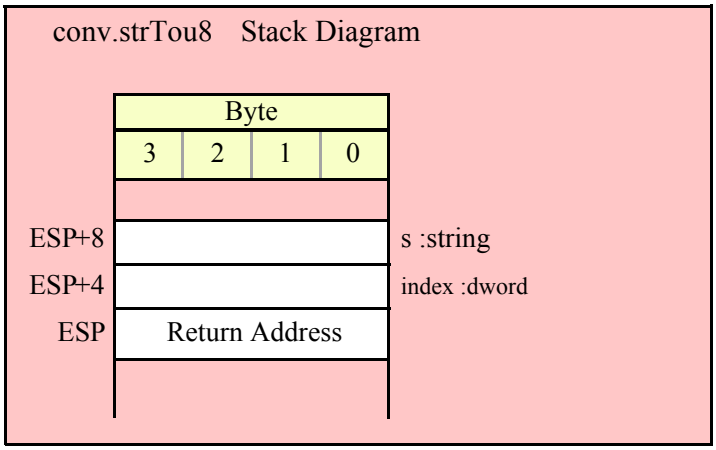


procedure conv.strTou8(s:string; index:dword)

This function converts the sequence of characters starting at zero-based offset index within the string parameter s to an 8-bit unsigned integer. It returns the value (in the range 0..255) in AL. Note that this function actually returns the zero-extended value in EAX, so you may use EAX if it is more convenient to do so.

```
  HLA high-level calling sequence examples:

  // The following will convert the characters at the beginning
  // of "decValueStr" to numeric form:
```

```
conv.strTou8( decValueStr, 0 );// Index=0 starts at beginning
mov( al, numericResult );


// The following demonstrates using a non-zero index:

conv.strTou8( "abc12", 3 ); // "12" begins at offset 3
mov( al, hex12 );


HLA low-level calling sequence examples:

  push( decValueStr );// Same as first example above
pushd( 0 );
call conv.strTou8;
mov( al, decNumericResult );


// Same as second example above

static
  str12 :string := "abc12";
  .
  .
  .
push( str12 );// Note that str12 points at "abc12".
pushd( 3 );// Index to "12" in "abc12".
call conv.strTou8;
mov( al, dec12 );
```

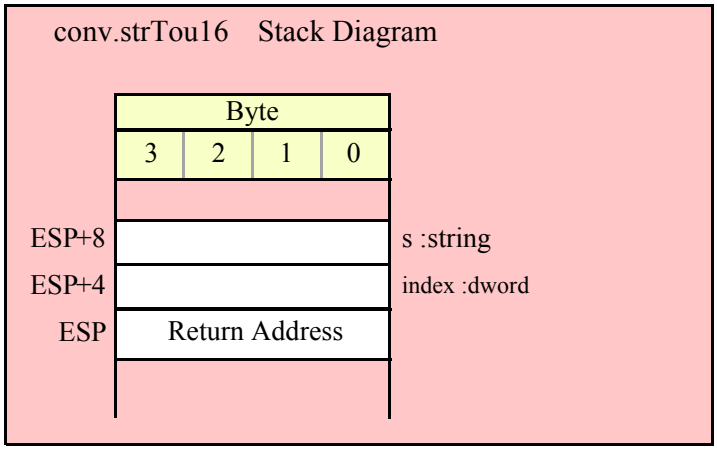


procedure conv.strTou16(s:string; index:dword)

This function converts the sequence of characters starting at zero-based offset index within the string parameter s to a 16-bit unsigned integer. It returns the value (in the range 0..65535) in AX. Note that this function actually returns the zero-extended value in EAX, so you may use EAX if it is more convenient to do so.

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

```
// The following will convert the characters at the beginning
// of "decValueStr" to numeric form:

conv.strTou16( hexValueStr, 0 );// Index=0 starts at beginning
mov( ax, wordVar );


// The following demonstrates using a non-zero index:

conv.strTou16( "abc1234", 3 ); // "1234" begins at offset 3
mov( ax, wordVar );


HLA low-level calling sequence examples:

  push( decValueStr );// Same as first example above
pushd( 0 );
call conv.strTou16;
mov( ax, wordVar );


// Same as second example above

static
  str1200 :string := "abc1200";
  .
  .
  .
push( str1200 );// Note that str1200 points at "abc1200".
pushd( 3 );// Index to "1200" in "abc1200".
call conv.strTou16;
mov( ax, wordVar );
```

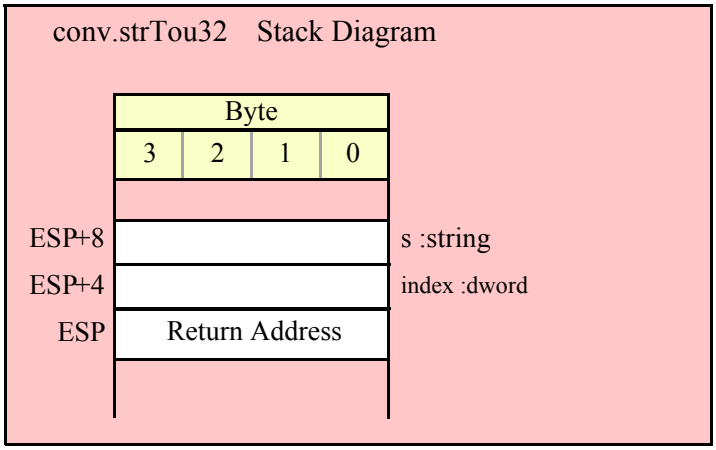


procedure conv.strTou32(s:string; index:dword)

This function converts the sequence of characters starting at zero-based offset index within the string parameter s to a 32-bit unsigned integer. It returns the value (in the range 0..4294967295) in EAX.

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

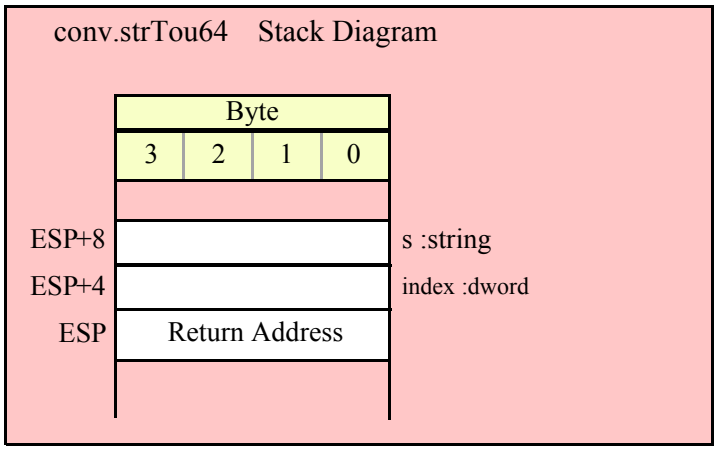
```
// The following will convert the characters at the beginning
// of "decValueStr" to numeric form:

conv.strTou32( decValueStr, 0 );// Index=0 starts at beginning
mov( eax, dwordVar );


// The following demonstrates using a non-zero index:

conv.strTou32( "abc12_345", 3 ); // "12_345" begins at offset 3
mov( eax, dwordVar );


HLA low-level calling sequence examples:

  push( decValueStr );// Same as first example above
pushd( 0 );
call conv.strTou32;
mov( eax, dwordVar );


// Same as second example above

static
  str12345 :string := "abc012_345";
  .
  .
  .
push( str12345 );// Note that str12345 points at "abc-12_345".
pushd( 3 ); // Index to "012_345" in "abc012_345".
call conv.strTou32;
mov( eax, dwordVar );// dwordVar now contains 12,345.
```



procedure conv.strTou64(s:string; index:dword)

This function converts the sequence of characters starting at zero-based offset index within the string parameter s to a 64-bit unsigned integer. It returns the value (in the range 0.. 18446744073709551615) in EDX:EAX (EDX contains the H.O. dword).

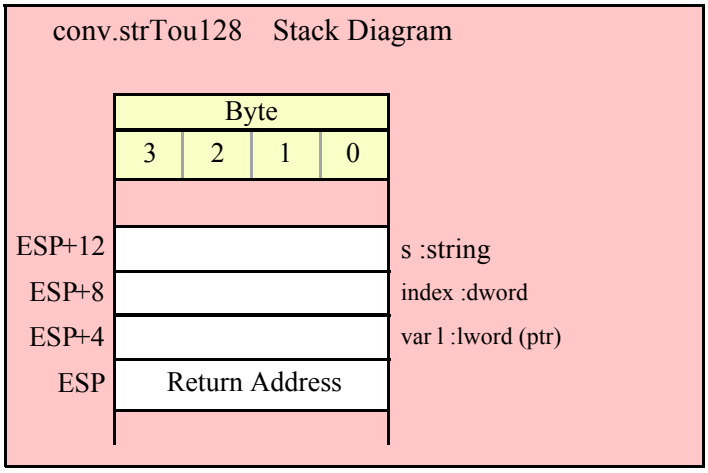
```
HLA high-level calling sequence examples:

// The following will convert the characters at the beginning
// of "decValueStr" to numeric form:

conv.strTou64( decValueStr, 0 );// Index=0 starts at beginning
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );


// The following demonstrates using a non-zero index:

conv.strTou64( "a9123", 1 ); // "9123" begins at offset 1
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );


HLA low-level calling sequence examples:

  push( decValueStr );// Same as first example above
pushd( 0 );
call conv.strTou64;
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );


// Same as second example above

static
  strabc12 :string := "a9123";
  .
  .
  .
push( strabc12 );// Note that strabc12 points at "a9123".
pushd( 1 ); // Index to "-123" in "a9123".
call conv.strTou64;
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );
```



procedure conv.strTou128(s:string; index:dword; var dest:lword)

This function converts the sequence of characters starting at zero-based offset index within the string parameter s to a 128-bit unsigned integer. It returns the value (in the range 0.. 340282366920938463463374607431768211456) in the parameter l that you pass by reference to this function.

```
HLA high-level calling sequence examples:

// The following will convert the characters at the beginning
// of "decValueStr" (index=0) to numeric form and store the
// 128-bit result into the 1wordDest variable:

conv.strTou128( decValueStr, 0, 1wordDest );


// The following demonstrates using a non-zero index:

conv.strTou128( "abc12345678901234567890", 3, 1wordDest );



HLA low-level calling sequence examples:

  //  Option #1: lwordDest is a STATIC/READONLY/STORAGE
  // variable:

  push( decValueStr );// Same as first example above
pushd( 0 );
pushd( &lwordDest );
call conv.strTou128;

  // Option #2: lwordDest is not a static object and
  // a 32-bit register is available for use:

  push( decValueStr );// Same as first example above
pushd( 0 );
lea( eax, lwordDest ); // Assuming EAX is available
push( eax );
call conv.strTou128;

  // Option #3: lwordDest is an automatic (var) object and
  // no 32-bit registers are available for use:

  push( decValueStr );// Same as first example above
pushd( 0 );
push( ebp );
add( @offset( lwordDest ), (type dword [esp]) );
call conv.strTou128;

  // Option #4: lwordDest is a complex addressing mode object and
  // no 32-bit registers are available for use:

  push( decValueStr );// Same as first example above
pushd( 0 );
sub( 4, esp );
push( eax );
lea( eax, lwordDest );
mov( eax, [esp+4] );
pop( eax );
call conv.strTou128;
```



8.6 Floating Point Conversions

These functions convert betweeen the three IEEE/Intel floating point formats and their string representation. 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 format.

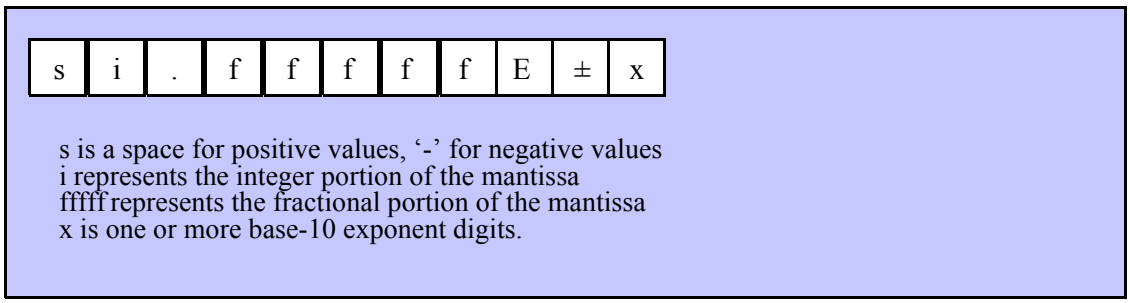
Note that the floating-point conversions do not insert underscores into the character sequences they produce. Therefore, these conversion functions ignore the internal underscores flag setting. If you wish to produce floating-point strings that have underscores between certain sets of digits, you should call one of these floating-point routines to do the basic conversion and then using other standard library routines to insert those underscores (or other character of your choosing) into the string.

FPU Note: The floating-point routines make use of the 80x86 x87 floating-point unit (FPU). Whenever you call one of these conversion routines, you must ensure that the CPU is operating in FPU mode (rather than MMX mode). If this is not the case, you should exit the MMX mode by executing an EMMS instruction prior to calling any of these conversion routines. As a general rule, you should use the SSE instructions rather than the MMX instructions and leave the CPU in FPU mode. Also note that the floating-point conversion routines make use of the FPU stack (probably as many as three elements, or so, just a guess) so you shouldn't leave any pending operations on the FPU stack when calling these conversion routines.

8.6.1 Exponential Floating-Point Conversions

The exponential floating-point conversion routines include conv.e32ToBuf, conv.e64ToBuf, conv.e80ToBuf, conv.e32ToStr, conv.e64ToStr, conv.e80ToStr, conv.a_e32ToStr, conv.a_e64ToStr, and conv.a_e80ToStr. The *Buf routines write the converted sequence of characters to memory, starting at the location pointed at by EDI. The *Str routines store the converted character data into a string object.

The routines convert their values to a character sequence using scientific (exponential) notation. These routines each at least two parameters: the value to convert and the field width of the result that control how these functions format the output These routines produce a string with the following format:



The width parameter specifies the exact number of print positions the value will consume (i.e., the length of the resulting string). The first position holds the sign of the value. This is a space for positive values or a minus sign for negative values. If you do not want a leading space in front of positive values you can either store a "+" over the top of this space character (if the number is zero or positive) or you can call str.trim to remove any leading space.

The exponent field ('x') always uses the minimum number of digits to exactly represent the exponent. If the exponent is non-negative, then these routines preface the value with a '+'. If the exponent is negative, then these functions preface the exponent value with a '-' character.

The minimum field width you should specify is five. This allows one print position for the leading sign character, one digit for the mantissa, the "E", the exponent sign, and one exponent digit. Obviously, values greater than 1E+9 or less than 1E-9 will require additional print positions to handle the additional exponent digits.

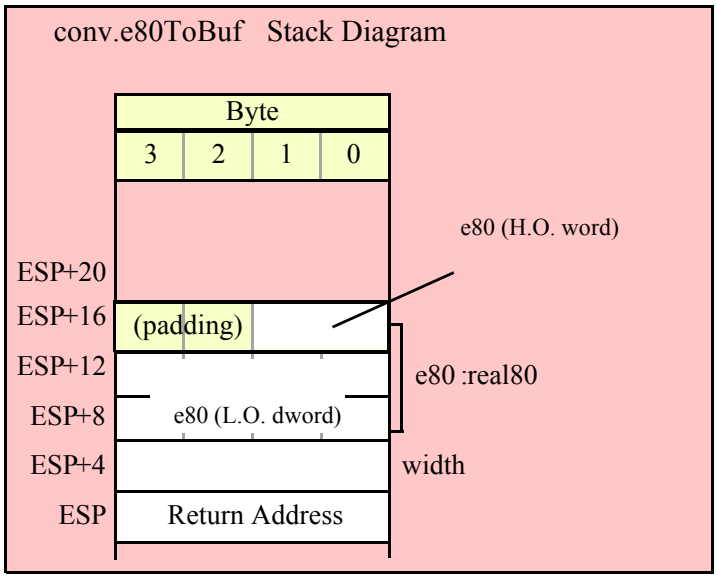
The number of fractional digits this routines produce is "(width - 5) - # exponent digits". So you should choose your width according to the expected exponent size and the number of digits you would like to have to the right of the decimal point.

8.6.2 Floating Point Numeric to Buffer Conversions, Exponential Form

The floating point numeric to buffer conversion routines, conv.e32ToBuf, conv.e64ToBuf, and conv.e80ToBuf, translate the three different binary floating point formats to a sequence of characters that they store into sequential memory locations starting at the address held in the EDI register. They return EDI pointing at the first byte beyond the converted sequence. Note that these functions do not zero terminate the string; if you want a zero-terminated string, then store a zero at the byte pointed at by EDI upon return from these functions.

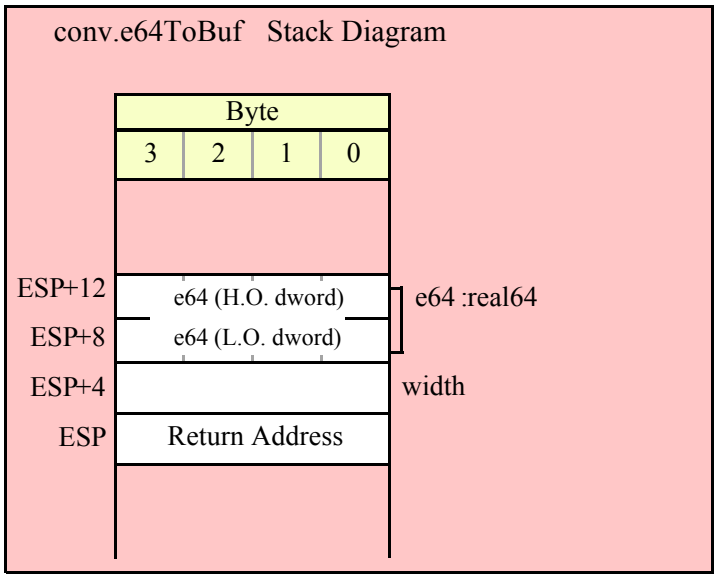
```
procedure conv.e80ToBuf
(
        e80:    real80;
        width:  uns32;
    var buffer: var in EDI
)
```

This function converts the 80-bit extended precision e80 value to its character representation using exponential/scientific notation. This function stores the resulting conversion starting at the address specified in EDI. It is the caller's responsibility to ensure that sufficient memory is avaiable at this starting address. On return, EDI will point at the first byte beyond the convered string. Note that the 80-bit extended precision format supports approximately 18 decimal digits of precision. Therefore, any digits beyond the 18th significant digit will contain garbage. Hence, your choice of width should not produce more than 18 mantissa digits.



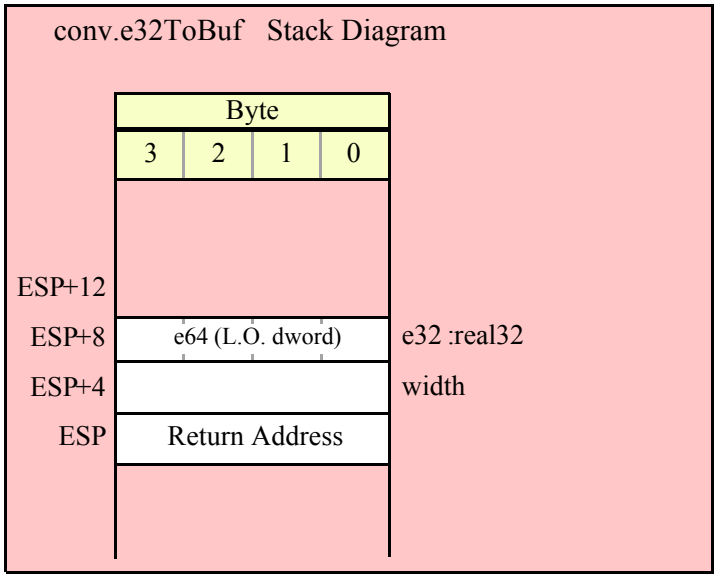
```
procedure conv.e64ToBuf
(
        e64:    real64;
        width:  uns32;
    var buffer: var in EDI
)
```

This function converts the 64-bit extended precision e64 value to its character representation using exponential/scientific notation. This function stores the resulting conversion starting at the address specified in EDI. It is the caller's responsibility to ensure that sufficient memory is avaiable at this starting address. On return, EDI will point at the first byte beyond the convered string. Note that the 64-bit extended precision format supports approximately 15 decimal digits of precision. Therefore, any digits beyond the 15th significant digit will contain garbage. Hence, your choice of width should not produce more than 15 mantissa digits.




```
procedure conv.e32ToBuf
(
        e32:    real32;
        width:  uns32;
    var buffer: var in EDI
)
```

This function converts the 32-bit extended precision e32 value to its character representation using exponential/scientific notation. This function stores the resulting conversion starting at the address specified in EDI. It is the caller's responsibility to ensure that sufficient memory is avaiable at this starting address. On return, EDI will point at the first byte beyond the convered string. Note that the 32-bit single precision format supports approximately 6-7 decimal digits of precision. Therefore, any digits beyond the seventh significant digit will contain garbage. Hence, your choice of width should not produce more than seven mantissa digits.

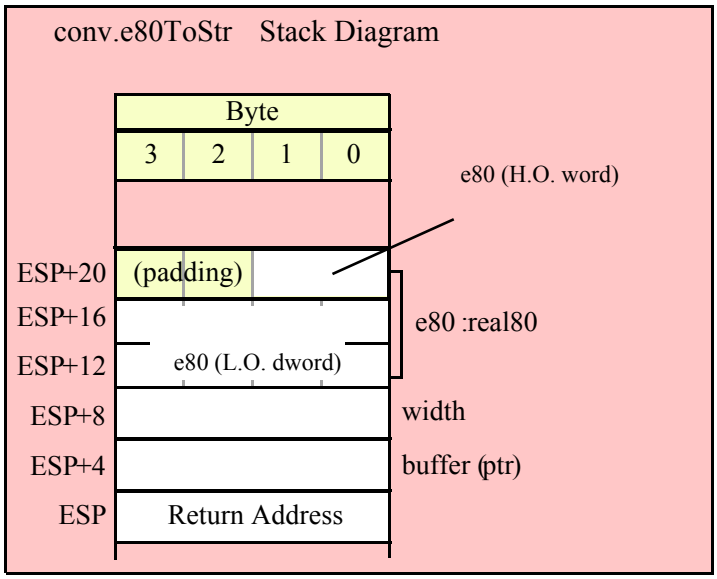


8.6.3 Floating Point Numeric to String Conversions, Exponential Form

The floating point numeric to string conversion routines translate the three different binary floating point formats to their string representation. The standard ("unadorned") functions store the string data into a string object that you pass as a parameter to the function. That string object must be preallocated and large enough to receive the string result (else a string overflow occurs). The "adorned" functions, whose names begin with "a_" automatically allocate storage on the heap, store the converted string into that heap object, and then return a pointer to the newly allocated string in the EAX register (it is the caller's responsibility to free the storage when it is no longer needed).

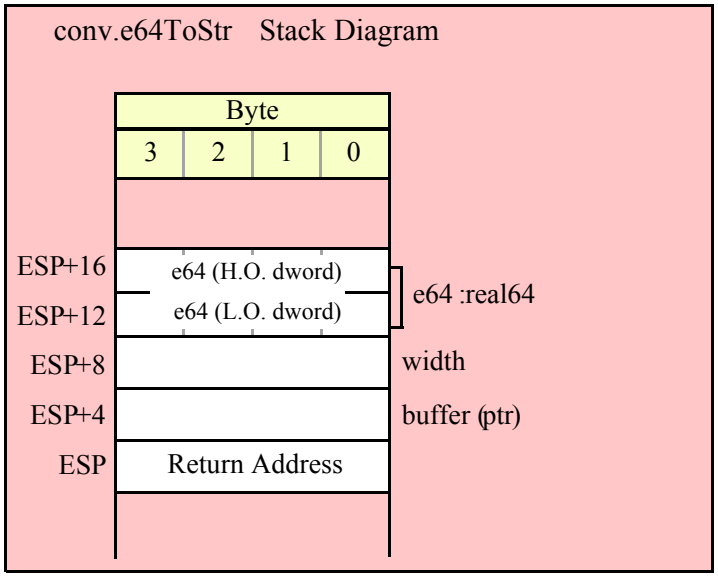
```
procedure conv.e80ToStr
(

    e80:        real80;
    width:      uns32;
    buffer:     string
)
```

This function converts the 80-bit extended precision e80 value to its string representation using exponential/scientific notation. This function stores the resulting string in the buffer variable whose MaxLength field must be at least width or this function will raise an exception. Note that the 80-bit extended precision format supports approximately 18 decimal digits of precision. Therefore, any digits beyond the 18th significant digit will contain garbage. Hence, your choice of width should not produce more than 18 mantissa digits.



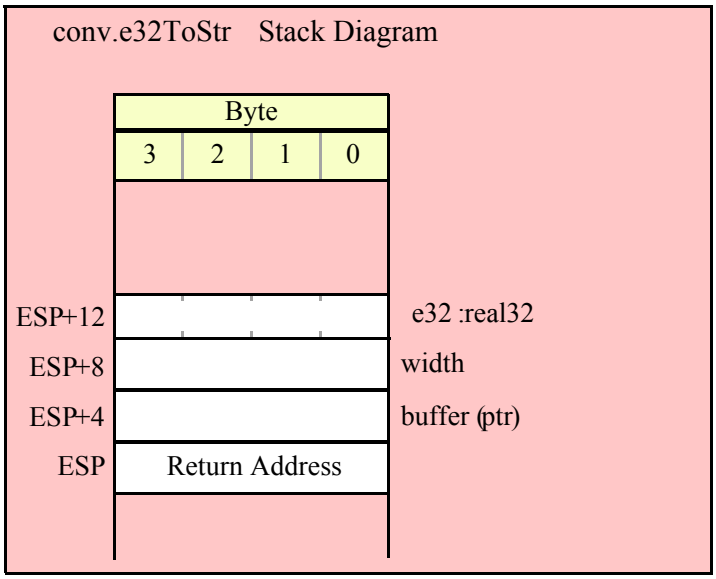

```
procedure conv.e64ToStr
(
     e64:        real64;
     width:      uns32;
     buffer:     string
)
```

This function converts the 64-bit double precision e64 value to its string representation using exponential/scientific notation. This function stores the resulting string in the buffer variable whose MaxLength field must be at least width or this function will raise an exception. Note that the 64-bit double precision format supports approximately 15 decimal digits of precision. Therefore, any digits beyond the 15th significant digit will contain garbage. Hence, your choice of width should not produce more than 15 mantissa digits.



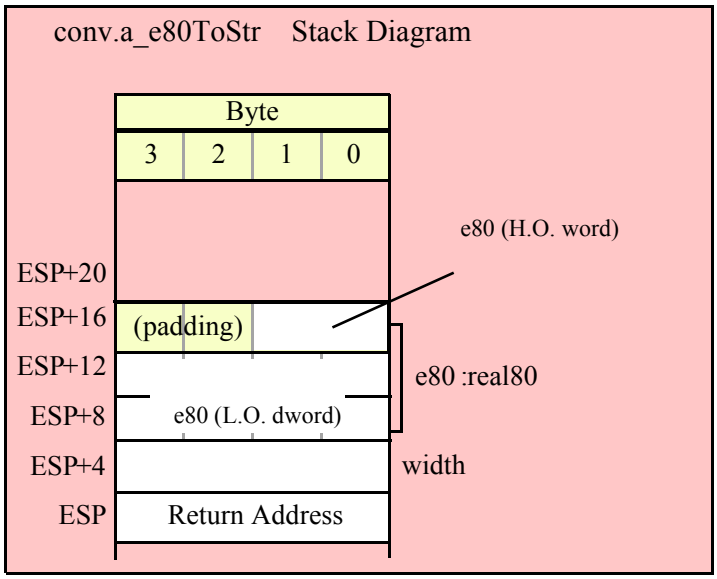
```
procedure conv.e32ToStr
(
     e32:  real32;
     width:uns32;
     buffer:string
)
```

This function converts the 32-bit single precision e32 value to its string representation using exponential/scientific notation. This function stores the resulting string in the buffer variable whose MaxLength field must be at least width or this function will raise an exception. Note that the 32-bit single precision format supports approximately 6-7 decimal digits of precision. Therefore, any digits beyond the seventh significant digit will contain garbage. Hence, your choice of width should not produce more than seven mantissa digits.



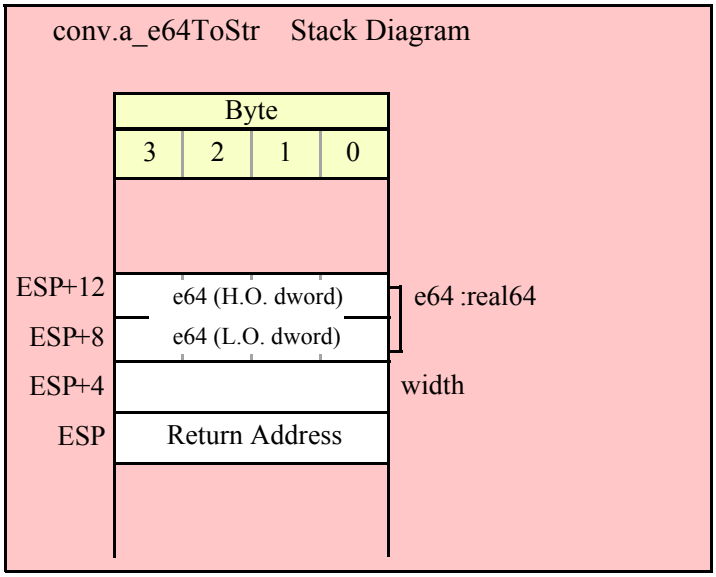

```
procedure conv.a_e80ToStr
(

     e80:        real80;
     width:      uns32
); @returns( "eax" );
```

This function converts the 80-bit extended precision e80 value to its string representation using exponential/scientific notation. This function stores the resulting string in storage it allocates on the heap and returns a pointer to that string in the EAX register. It is the caller's responsibility to deallocate this storage when they are done with it. Note that the 80-bit extended precision format supports approximately 18 decimal digits of precision. Therefore, any digits beyond the 18^{th} significant digit will contain garbage. Hence, your choice of width should not produce more than 18 mantissa digits.



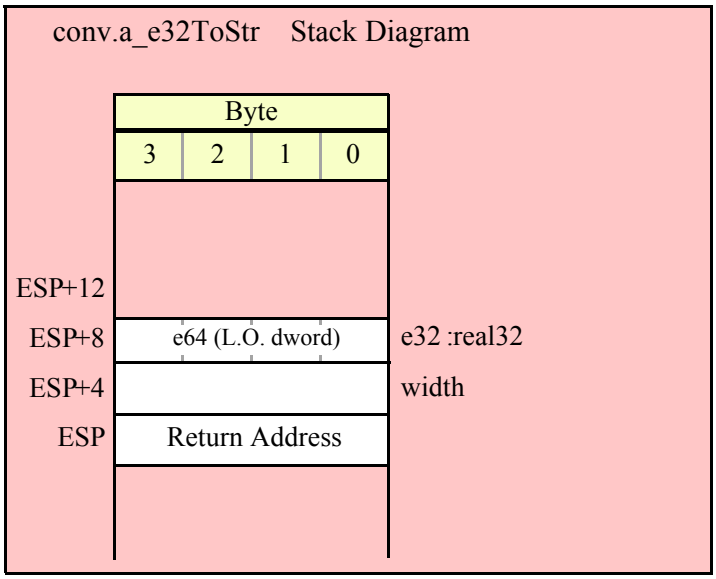

```
procedure conv.a_e64ToStr
(
     e64:         real64;
     width:       uns32
); @returns( "eax" );
```

This function converts the 64-bit double precision e64 value to its string representation using exponential/ scientific notation. This function stores the resulting string in storage it allocates on the heap and returns a pointer to that string in the EAX register. It is the caller's responsibility to deallocate this storage when they are done with it. Note that the 64-bit double precision format supports approximately 15 decimal digits of precision. Therefore, any digits beyond the 15th significant digit will contain garbage. Hence, your choice of width should not produce more than 15 mantissa digits.



```
procedure conv.a_e32ToStr
(
     e32:        real32;
     width:      uns32
); @returns( "eax" );
```

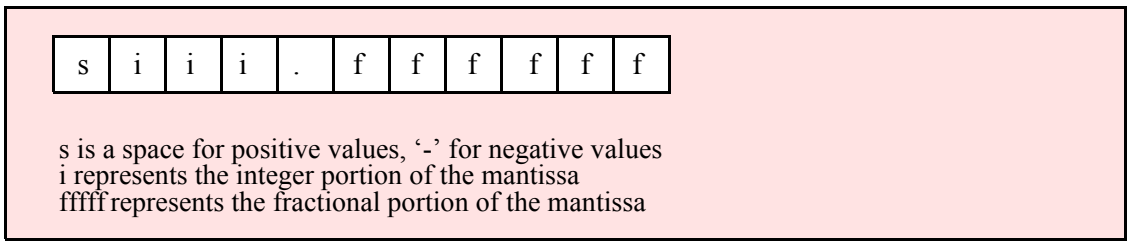
This function converts the 32-bit single precision e32 value to its string representation using exponential/scientific notation. This function stores the resulting string in storage it allocates on the heap and returns a pointer to that string in the EAX register. It is the caller's responsibility to deallocate this storage when they are done with it. Note that the 32-bit single precision format supports approximately 6-7 decimal digits of precision. Therefore, any digits beyond the seventh significant digit will contain garbage. Hence, your choice of width should not produce more than seven mantissa digits.



8.6.4 Floating Point Numeric to Character Conversions, Decimal Form

Although scientific (exponential) notation is the most general display format for real numbers, real numbers you display in this format are difficult to read. Therefore, the standard library conversions module also provides a set of functions that convert real values to their decimal string equivalent. Although you cannot (practically) use these decimal conversion routines for all real values, they are applicable to a wide variety of common numbers you will use in your programs.

These functions all have at least 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 convert their values to the following string format:



The width parameter specifies the length of the resulting string. This value must be less than or equal to the destination string's MaxLength value or these functions will raise an exception. The decimalpts parameter to these functions specify the number of digits to the right of the decimal point. If this parameter contains zero, then these functions display the value as an integer (no fractional digits and no decimal point). If this parameter is non-zero, then these routines produce the specified number of decimal digits along with a decimal point.

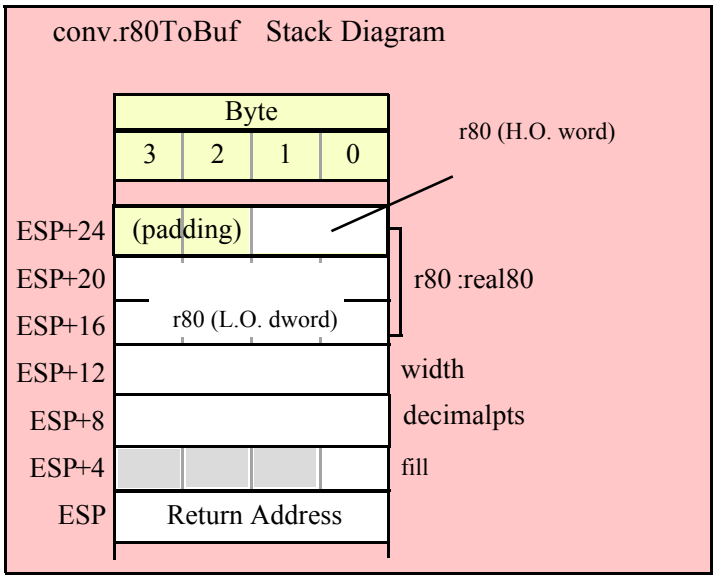
The width parameter specifies the total size of the resulting string. If decimalpts is zero, then the width value must be at least one greater than the number of digits that appear to the left of the decimal point (the extra position is for the sign character. If the decimalpts parameter is non-zero, then width must be at least (decimalpts + 2 + # integer digits). If width is not sufficiently large, then these functions produce a string containing width "#" characters to denote a conversion error.

If the width value is sufficiently large and the decimalpts sufficiently small then these routines will fill the extra print positions using the fill character you pass as a parameter. For example, if you convert the value -1.5 with a width of six, a decimalpts value of two, and a fill character of "*" these routines produce the string "*-1.50".

The floating point numeric to buffer conversion routines, conv.r32ToBuf, conv.r64ToBuf, and conv.r80ToBuf, translate the three different binary floating point formats to a sequence of characters that they store into sequential memory locations starting at the address held in the EDI register. They return EDI pointing at the first byte beyond the converted sequence. Note that these functions do not zero terminate the string; if you want a zero-terminated string, then store a zero at the byte pointed at by EDI upon return from these functions.

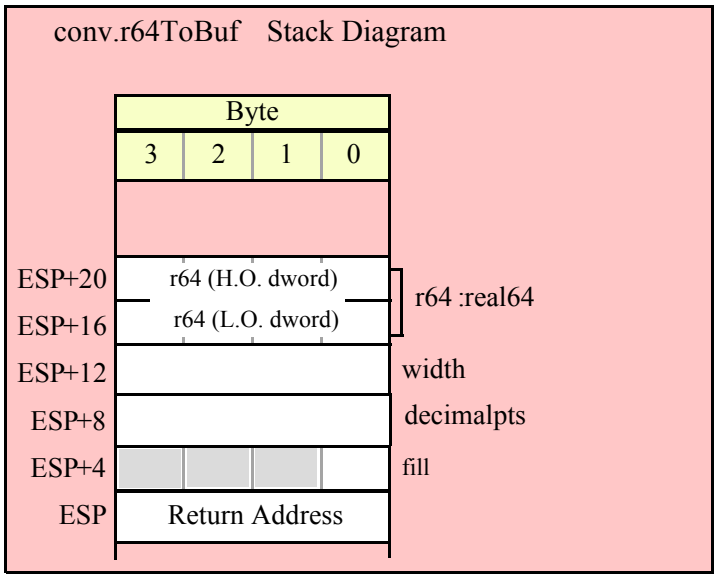
```
procedure conv.r80ToBuf
(
            r80:        real80;
            width:      uns32;
            decimalpts: uns32;
            fill:       char;
  var       buffer:     var in edi
)
```

This function converts the 80-bit extended precision r80 value to its string representation using decimal notation. This function stores the resulting character sequence into sequential memory locations starting at the address held in the EDI register. They return EDI pointing at the first byte beyond the converted sequence. Note that these functions do not zero terminate the string; if you want a zero-terminated string, then store a zero at the byte pointed at by EDI upon return from these functions. Note that the 80-bit single precision format supports approximately 18 decimal digits of precision. Therefore, any digits beyond the 18^{th} significant digit will contain garbage. Hence, your choice of width should not produce more than 18 mantissa digits. Do keep in mind that the average person has trouble comprehending value with more than six or seven digits. For values that are routinely outside this range you may want to use exponential form to display the number with a limited number of significant digits.



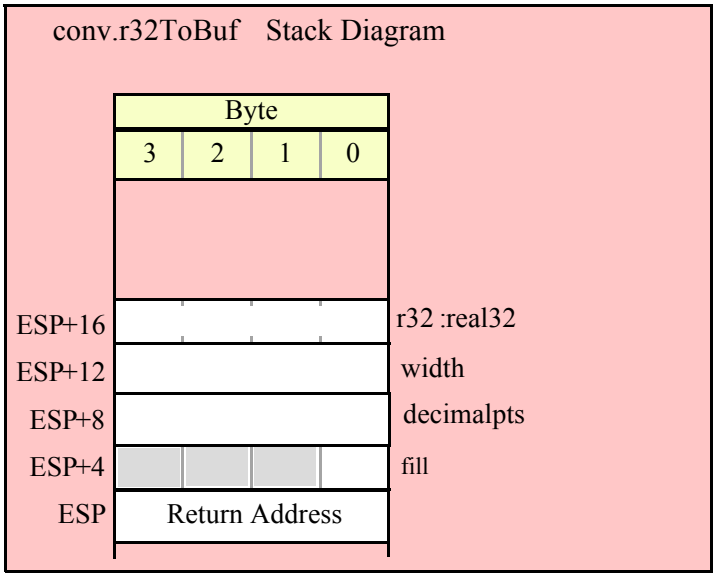

```
procedure conv.r64ToBuf
(
            r64:        real64;
            width:      uns32;
            decimalpts: uns32;
            fill:       char;
  varbuffer:            var in edi
)
```

This function converts the 64-bit double precision r64 value to its string representation using decimal notation. This function stores the resulting character sequence into sequential memory locations starting at the address held in the EDI register. They return EDI pointing at the first byte beyond the converted sequence. Note that these functions do not zero terminate the string; if you want a zero-terminated string, then store a zero at the byte pointed at by EDI upon return from these functions. Note that the 64-bit single precision format supports approximately 15 decimal digits of precision. Therefore, any digits beyond the 15^{th} significant digit will contain garbage. Hence, your choice of width should not produce more than 15 mantissa digits. Do keep in mind that the average person has trouble comprehending value with more than six or seven digits. For values that are routinely outside this range you may want to use exponential form to display the number with a limited number of significant digits.




```
procedure conv.r32ToBuf
(
            r32:        real32;
            width:      uns32;
            decimalpts: uns32;
            fill:       char;
  varbuffer:            var in edi
)
```

This function converts the 32-bit single precision r32 value to its string representation using decimal notation. This function stores the resulting Note that the 32-bit single precision format supports approximately 6-7 decimal digits of precision. Therefore, any digits beyond the seventh significant digit will contain garbage. Hence, your choice of width should not produce more than seven mantissa digits.

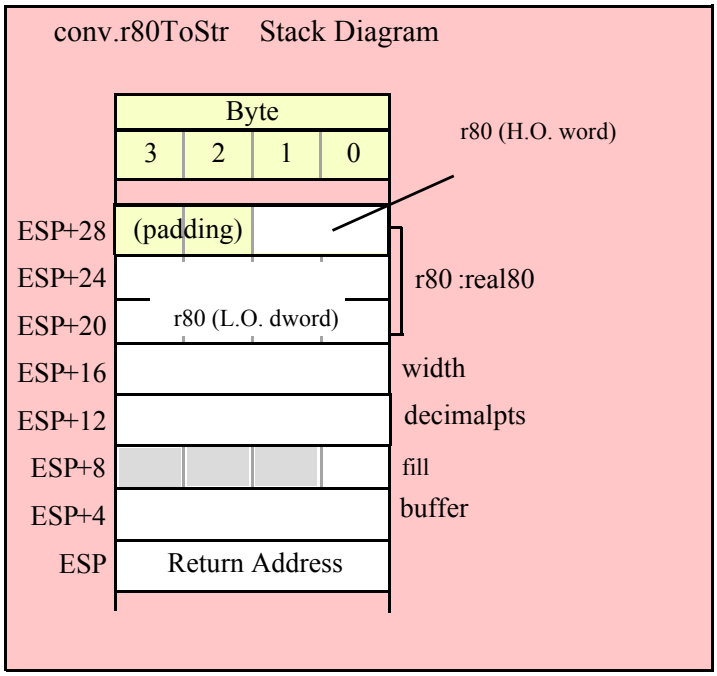


8.6.5 Floating-Point Numeric to String Conversions, Decimal Form

The floating point numeric to string conversion routines translate the three different binary floating point formats to their string representation. The standard ("unadorned") functions store the string data into a string object that you pass as a parameter to the function. That string object must be preallocated and large enough to receive the string result (else a string overflow occurs). The "adorned" functions, whose names begin with "a_" automatically allocate storage on the heap, store the converted string into that heap object, and then return a pointer to the newly allocated string in the EAX register (it is the caller's responsibility to free the storage when it is no longer needed).

```
procedure conv.r80ToStr
(

     r80:        real80;
     width:      uns32;
     decimalpts: uns32;
     fill:       char;
     buffer:     string
)
```

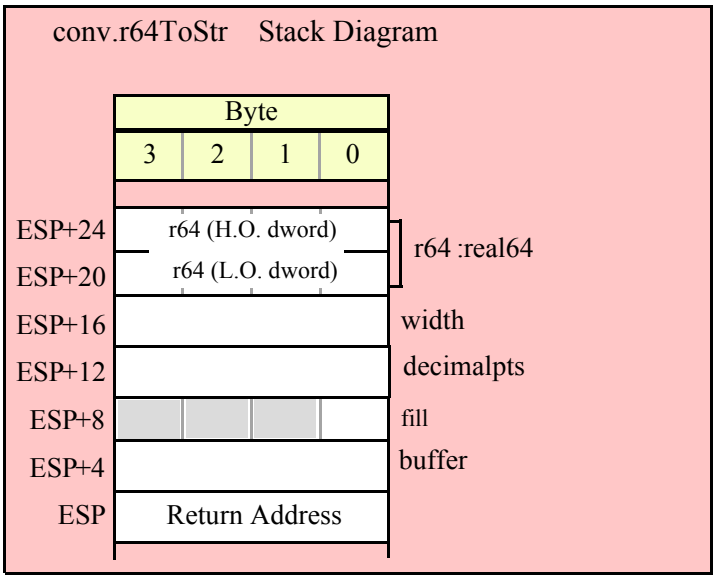
This function converts the 80-bit extended precision r80 value to its string representation using decimal notation. This function stores the resulting string in the buffer variable whose MaxLength field must be at least width or this function will raise an exception. Note that the 80-bit single precision format supports approximately 18 decimal digits of precision. Therefore, any digits beyond the 18th significant digit will contain garbage. Hence, your choice of width should not produce more than 18 mantissa digits. Do keep in mind that the average person has trouble comprehending value with more than six or seven digits. For values that are routinely outside this range you may want to use exponential form to display the number with a limited number of significant digits.




```
procedure conv.r64ToStr
(
     r64:        real64;
     width:      uns32;
```

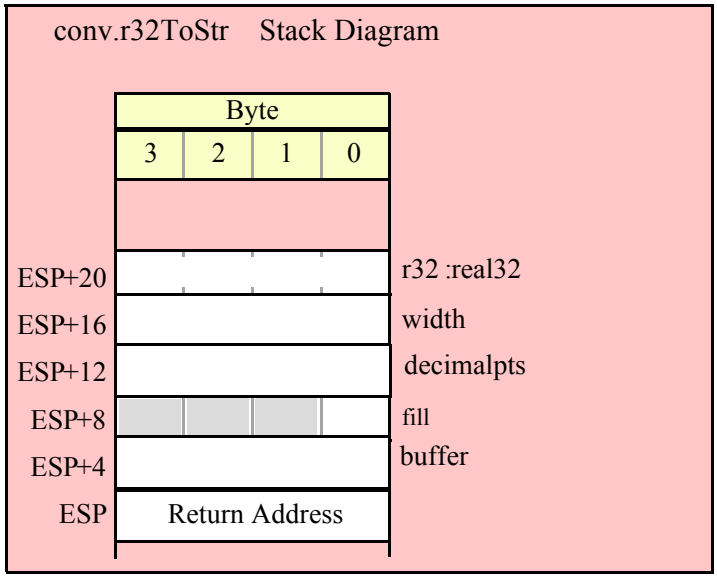
```
    decimalpts: uns32;
    fill:       char;
    buffer:     string
)
```

This function converts the 64-bit double precision r64 value to its string representation using decimal notation. This function stores the resulting string in the buffer variable whose MaxLength field must be at least width or this function will raise an exception. Note that the 64-bit single precision format supports approximately 15 decimal digits of precision. Therefore, any digits beyond the 15th significant digit will contain garbage. Hence, your choice of width should not produce more than 15 mantissa digits. Do keep in mind that the average person has trouble comprehending value with more than six or seven digits. For values that are routinely outside this range you may want to use exponential form to display the number with a limited number of significant digits.



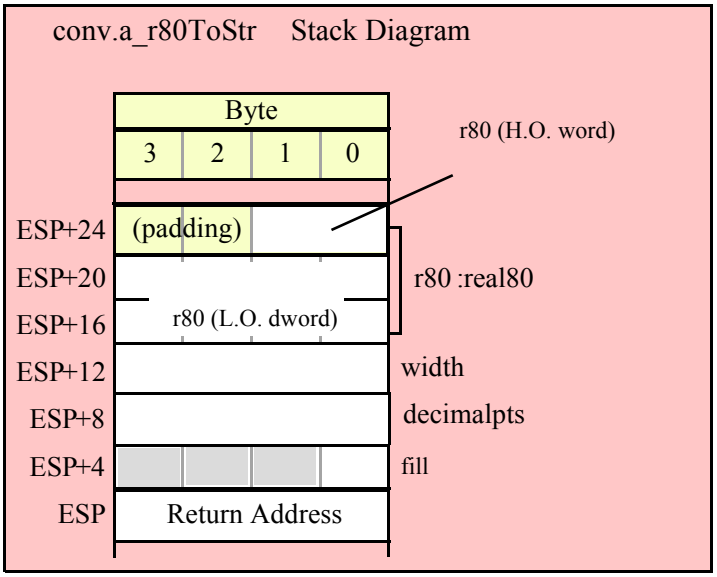

```
procedure conv.r32ToStr
(
    r32:        real32;
    width:      uns32;
    decimalpts: uns32;
    fill:       char;
    buffer:     string
)
```

This function converts the 32-bit single precision r32 value to its string representation using decimal notation. This function stores the resulting string in the buffer variable whose MaxLength field must be at least width or this function will raise an exception. Note that the 32-bit single precision format supports approximately 6-7 decimal digits of precision. Therefore, any digits beyond the seventh significant digit will contain garbage. Hence, your choice of width should not produce more than seven mantissa digits.



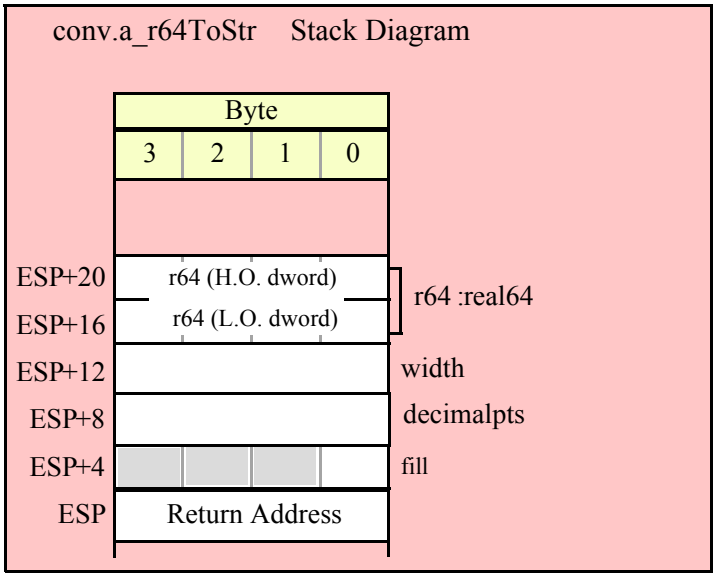

```
procedure conv.a_r80ToStr
(
     r80:         real80;
     width:       uns32;
     decimalpts: uns32;
     fill:        char
); @returns( "eax" );
```

This function converts the 80-bit extended precision r80 value to its string representation using decimal notation. This function stores the resulting string in storage it allocates on the heap and returns a pointer to that string in the EAX register. It is the caller's responsibility to deallocate this storage when they are done with it.. Note that the 80-bit single precision format supports approximately 18 decimal digits of precision. Therefore, any digits beyond the 18th significant digit will contain garbage. Hence, your choice of width should not produce more than 18 mantissa digits. Do keep in mind that the average person has trouble comprehending value with more than six or seven digits. For values that are routinely outside this range you may want to use exponential form to display the number with a limited number of significant digits.



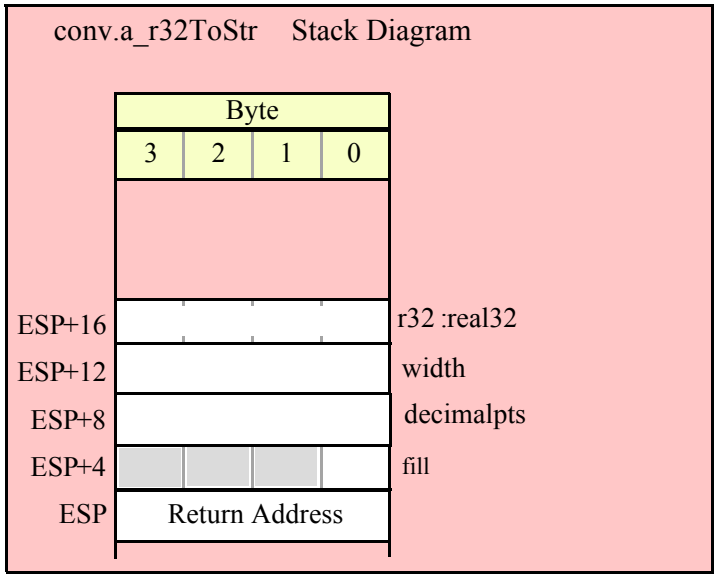
```
procedure conv.a_r64ToStr
(
     r64:        real64;
     width:      uns32;
     decimalpts: uns32;
     fill:       char
); @returns( "eax" );
```

This function converts the 64-bit double precision r64 value to its string representation using decimal notation. This function stores the resulting string in storage it allocates on the heap and returns a pointer to that string in the EAX register. It is the caller's responsibility to deallocate this storage when they are done with it. Note that the 64-bit single precision format supports approximately 15 decimal digits of precision. Therefore, any digits beyond the 15th significant digit will contain garbage. Hence, your choice of width should not produce more than 15 mantissa digits. Do keep in mind that the average person has trouble comprehending value with more than six or seven digits. For values that are routinely outside this range you may want to use exponential form to display the number with a limited number of significant digits.




```
procedure conv.a_r32ToStr
(
     r32:        real32;
     width:      uns32;
     decimalpts: uns32;
     fill:       char
); @returns( "eax" );
```

This function converts the 32-bit single precision r32 value to its string representation using decimal notation. This function stores the resulting string in storage it allocates on the heap and returns a pointer to that string in the EAX register. It is the caller's responsibility to deallocate this storage when they are done with it. Note that the 32-bit single precision format supports approximately 6-7 decimal digits of precision. Therefore, any digits beyond the seventh significant digit will contain garbage. Hence, your choice of width should not produce more than seven mantissa digits.



8.6.6 Floating Point String/Buffer to Numeric Conversions

The floating-point string to numeric routines convert characters found in a character sequence to an 80-bit IEEE floating-point format. There are two versions – conv.atof and conv.strToFlt. conv.atof operates on an arbitrary sequence of characters in memory and conv.strToFlt operates on a string variable.

These routines will skip over any leading underscore and delimiter characters (specified by the internal delimiters character set, see the discussion of conv.setDelimiters and conv.getDelimiters for details). These functions will convert all characters in the sequence until encountering an illegal floating-point character (decimal digits, a decimal point, 'e' or 'E', and an optional sign for the exponent and mantissa). If the first non-acceptable character is not the end of string or a delimiter character, these functions will raise a conversion exception. If the character is not a valid 7-bit ASCII character, these functions will raise an illegal character exception.

The conv.strToFlt functions has two parameters: a string object and an index into that string. Numeric conversion because at the zero-based character position specified by the index parameter. For example, the invocation

```
conv.strToFlt( someStr, 5 );
```

begins the conversion starting with the sixth character (index 5) in someStr. These functions will raise an "index out of range" exception if the supplied index is greater than the size of the string the first parameter specifies. They will return a null pointer reference exception if the string parameter is NULL (they will return an illegal memory access exception if the first parameter is not a valid pointer).

These functions always convert their strings to an 80-bit floating-point value and leave that value sitting on the top of the FPU stack (ST0). If you want the conversion to a 32-bit or 64-bit floating-point format, then use the fstp instruction to store the result in whatever destination format you desire (real32, real64, or real80).

```
procedure conv.atof( bufptr: dword in esi ); @returns( "st0" );
```

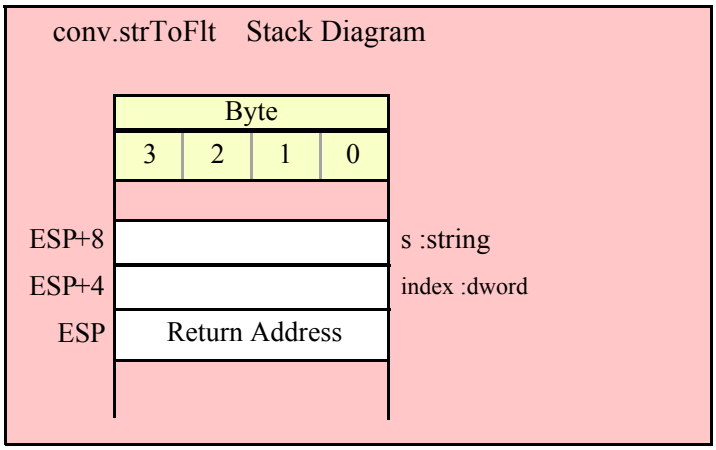
This routine assumes that ESI is pointing at a sequence of characters that represents a floating point number. The characters are converted to numeric form and the result is returned in ST0. ESI is left pointing at the first character beyond the converted characters. This function raises an exception if the value begins with something other than a standard numeric, '-', or delimiter character or ends with something other than a standard delimiter character (or the end of string).

This routine accepts floating point input in either decimal or exponential form.

```
procedure conv.strToFlt( s:string; index:dword );  @returns( "st0" );
```

This function converts the sequence of characters starting at position index in s to the equivalent extended precision floating point value and it leaves the result in ST0. This function raises an exception if the value begins with something other than a standard numeric, '-', or delimiter character, or ends with something other than a standard delimiter character (or the end of string).

This routine accepts floating point input in either decimal or exponential form.



8.6.7 Roman Numeral Conversion

```
procedure conv.roman( Arabic:uns32; rmn:string )
```

This procedure converts the specified integer value (Arabic) into a string that contains the Roman numeral representation of the value. Note that this routine only converts integer values in the range 1..3,999 to Roman numeral form. Since ASCII text doesn't allow overbars (that multiply roman digits by 1,000), this function doesn't handle really large Roman numbers. A different character set would be necessary for that.

```
conv.a_roman( Arabic:uns32 )
```

Just like the routine above, but this one allocates storage for the string and returns a pointer to the string in the EAX register.