15 The File I/O Module (fileio.hhf)

This unit contains routines that read data from and write data to files. The fileio functions can be broken down into four generic categories: general functions that open and close files, file position functions that get or set the current file position (or test the file position), output functions that write data to a file, and input functions that read data from a file.

Note to stdlib v1.x users: Several routines originally found in the fileio package have been moved to the new filesys package. The affected routines did not operate on file data, but on the file system itself. Examples include file deletion, get working directory, and change directory. Please see the filesys.rtf documentation for a description of those routines.

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

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

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

15.1 Conversion Format Control

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

When reading numeric data from a text file, the fileio functions use an internal delimiters character set to determine which characters may legally end a sequence of numeric digits. You can change the complexion of this character set using the conv.getDelimiters and conv.setDelimiters functions. Please refer to their documentation in the conv.rtf file for more details.

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

15.2 General File I/O Functions

Here are the file output routines provided by the HLA fileio unit:

Note: fileio.open is part of the os_fileio module in v2.0 of the HLA stdlib. This function has not been updated yet and the semantics may change during the conversion to v2.0.

```
fileio.open( FileName: string; Access:dword ); @returns( "eax" );
```

The fileio.open routine opens the file by the specified name. The Access parameter is one of the following:

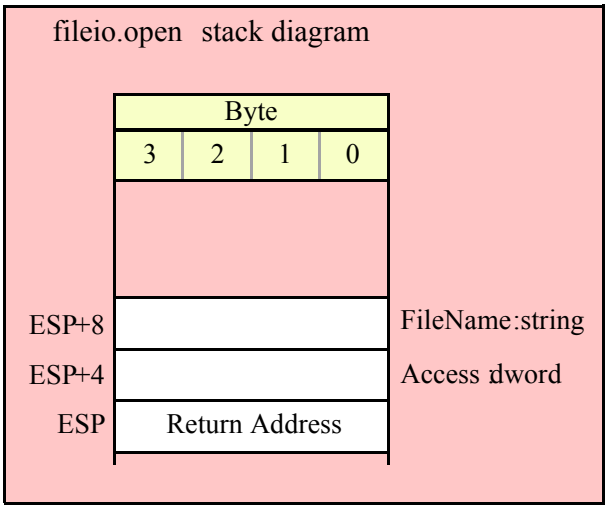
- fileio.r
- fileio.w
- fileio.rw
- fileio.a

The fileio.r constant tells HLA to open the file for read-only access. The fileio.w constant tells HLA to open the file for writing. Using the fileio.rw constant tells fileio.open to open the file for reading and writing. The fileio.a option tells the fileio.open function to open the file for writing and append all written data to the end of the file.

This routine raise an exception if there is a problem opening the file (e.g., the file does not exist). If the file is successfully opened, this function returns the file handle in the EAX register.

```
HLA high-level calling sequence examples:

fileio.open( "myfile.txt", fileio.r );
mov( eax, fileHandle );

// Note: the Access parameter is almost always a constant in
// calls to fileio.open. However, if you want to pass a variable
// value or a register value in this parameter, you may certainly
// do so:

fileio.open( filenameStr, accessVarByte );
mov( eax, fileHandle );

fileio.open( someStr, al );
mov( eax, fileHandle );


HLA low-level calling sequence examples:

// Constant Access value:

push( filenameStr );
pushd( fileio.r );
call fileio.open;
mov( eax, fileHandle );

// Access value in register (AL in this example)

push( filenameStr );
push( eax );
call fileio.open;
mov( eax, fileHandle );

// Access value is in a (byte) variable

push( filenameStr );
push( (type dword accessValue)); //Not always safe!
call fileio.open;
mov( eax, fileHandle );

// Solution if accessing accessValue as a dword
// might cause a memory access error (last three
// bytes on a 4K page in memory, etc.):

push( filenameStr );
sub( 4, esp );
push( eax );
movzx( accessValue, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.open;
mov( eax, fileHandle );

// Note: If you want to use a string literal, the best solution is
// to create a string object in the readonly section, e.g.,
//
// readonly
//   filenameStr :string := "myfile.txt";
//
// and just use the "filenameStr" object you've created. You may also
```

```
// do the following if you have a register available:

lea( eax, "myfile.txt" );
push( eax );
pushd( fileio.r );
call fileio.open;
mov( eax, fileHandle );
```

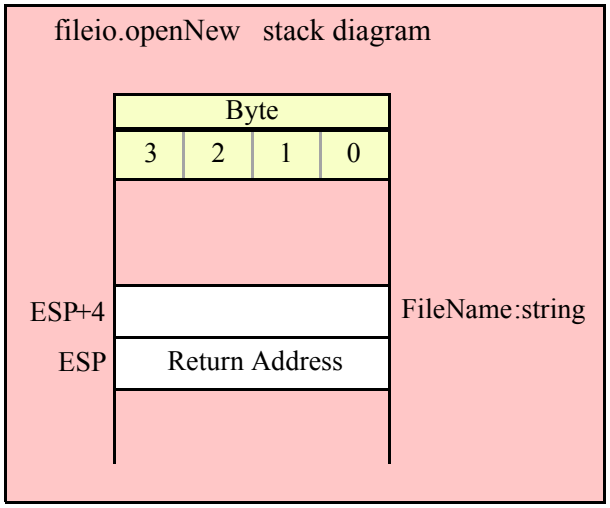


`fileio.openNew( FileName: string ); @returns( "eax" );`

This function opens a new file for writing. The single parameter specifies the file's (path) name. This function raises an exception if there is an error opening the file. If the file is opened successfully, this function returns the file handle in the EAX register. If the file already exists, this function will successfully open the file and delete any existing data in the file.

```
HLA high-level calling sequence examples:

fileio.openNew( "myfile.txt" );
mov( eax, fileHandle );

// If the filename string pointer is in a register (EAX):

fileio.openNew( eax );
mov( eax, fileHandle );


HLA low-level calling sequence examples:


push( filenameStr );
call fileio.openNew;
mov( eax, fileHandle );

// If the string pointer value is in a register (EAX
// in this example):

push( eax );
call fileio.openNew;
mov( eax, fileHandle );
```

```
// Note: If you want to use a string literal, the best solution is
// to create a string object in the readonly section, e.g.,
//
// readonly
//   filenameStr :string := "myfile.txt";
//
// and just use the "filenameStr" object you've created. You may also
// do the following if you have a register available:

lea( eax, "myfile.txt" );
push( eax );
call fileio.openNew;
mov( eax, fileHandle );
```

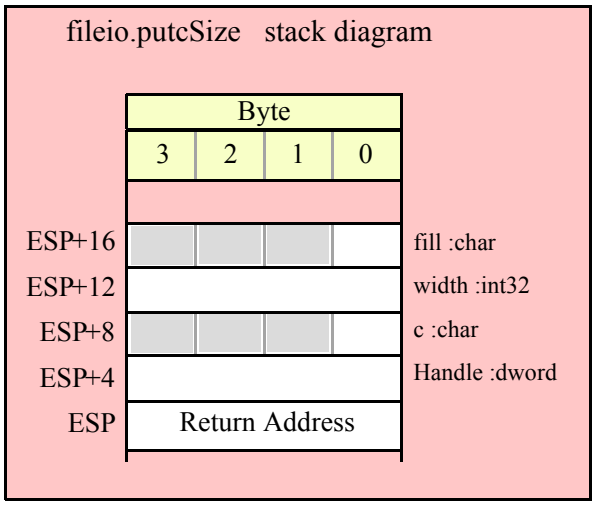


`fileio.close( Handle:dword );`

This function closes the file specfied by the handle passed as the parameter. You should close all files as soon as you are done using them. Note that successful program termination automatically closes all files, but it is exceeding poor programming practice to rely on the operating system to close any files you've left open. Were the machine to crash, data could be lost; for this reason, you should close all files as soon as you are finished reading and writing data.

```
HLA high-level calling sequence examples:

fileio.close( fileHandle );

// If the file handle is in a register (EAX):

fileio.close( eax );


HLA low-level calling sequence examples:

push( fileHandle );
call fileio.close;

// If the file handle is in a register (EAX):
```

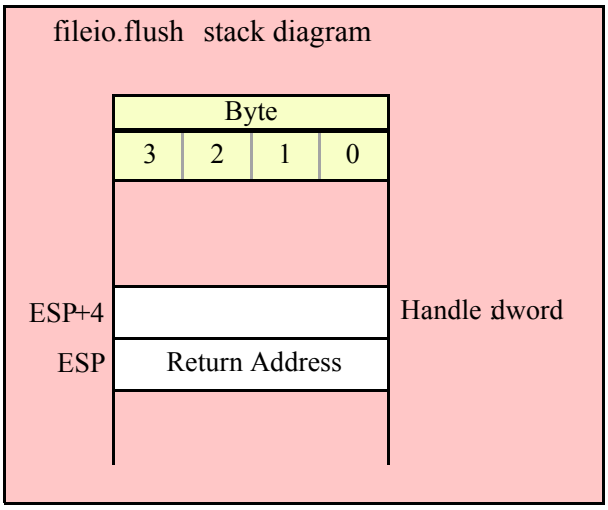
```
push( eax );
call fileio.close;
```



fileio.flush(Handle:dword);

This function flushes all pending data to the file (same operation as closing the file, without actually closing it). Note that successful program termination automatically closes all files, but were a crash to occur, some data might be lost. Flushing the file on a periodic basis can help prevent file data loss.

```
HLA high-level calling sequence examples:

fileio.flush( fileHandle );

// If the file handle is in a register (EAX):

fileio.flush( eax );


HLA low-level calling sequence examples:


push( fileHandle );
call fileio.flush;

// If the file handle is in a register (EAX):

push( eax );
call fileio.flush;
```



`fileio.eof( Handle:dword ); @returns( "al" );`

This function returns true (1) in AL if the specified file is at the end of file. It returns false (0) otherwise. Note that this function actually returns true/false in EAX even though the "returns" value is "AL". So don't count on it preserving the value in AH or the upper 16 bits of EAX.

Warning: fileio.eof only functions properly for actual disk files. If you attempt to read data from an interactive device like the system console (keyboard) or a serial port, fileio.eof's behavior is incorrect (it will wind up eating a character from the interactive input stream every time you call it). Unfortunately, neither Windows nor Linux provides a way to test for EOF until after you've actually read a character from the input stream. A better solution, which works fine with both interactive input streams and file data is to use HLA's try..endtry statement to trap and EOF error when it occurs. For example, rather than writing the following:

```
while( !fileio.eof( someHandle )) do
  .
  .
  .
endwhile;
```

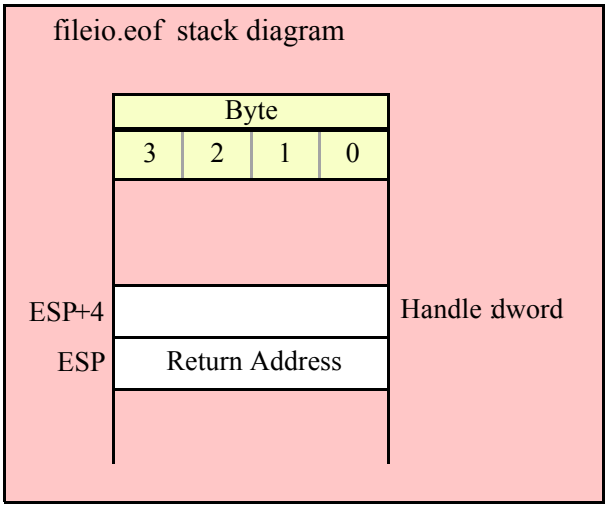
You should write the following:

```
try
  forever
      .
      .
      .
  endfor;
  exception( ex.EndOfFile );

endtry;
```

Note: under Windows, fileio.eof always returns false for character device files (e.g., keyboard input) and it returns false for all other non-disk file device types. Note that if the user presses ctrl-Z on the keyboard, fileio.eof will not return true, but the system will return an ex.endOfFile exception. If there is any chance you'll be reading data from a device file rather than a disk file, always use the try..endtry block to test for EOF.

```
HLA high-level calling sequence examples:

while( !fileio.eof( fileHandle ) ) do

  <<something while not at EOF>>
```

```
endwhile;


HLA low-level calling sequence examples:

whileNotEOF:
  push( fileHandle );
  call fileio.flush;
  cmp( al, true );
  jne atEOF;


     << something while not at EOF>>

  jmp whileNotEOF;
atEOF:
```

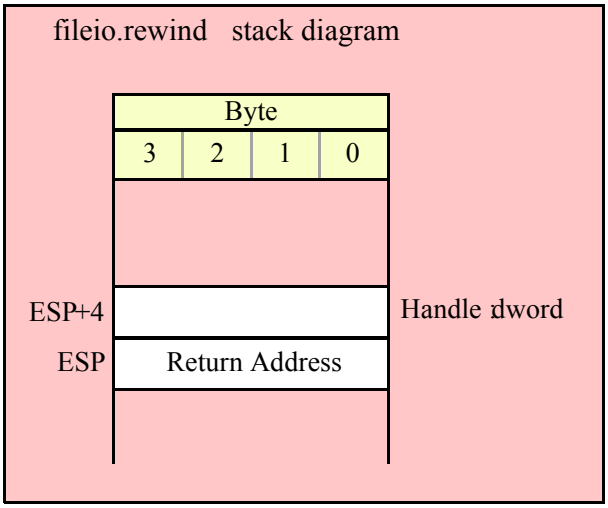


`fileio.rewind( Handle:dword ); @returns( "eax" );`

The Handle parameter specifies the handle of an open file. This function positions the file pointer to the beginning of the file (file position zero). This function returns the error code in EAX.

```
HLA high-level calling sequence examples:

fileio.rewind( fileHandle );

// If the file handle is in a register (EAX):

fileio.rewind( eax );


HLA low-level calling sequence examples:


push( fileHandle );
call fileio.rewind;

// If the file handle is in a register (EAX):

push( eax );
```

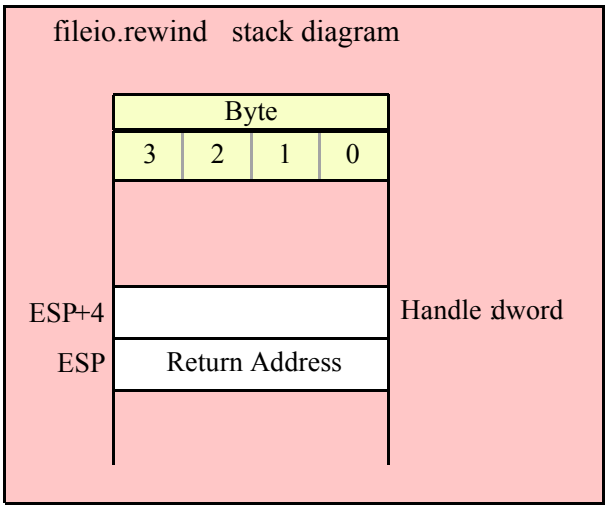
```
call fileio.rewind;
```



`fileio.append( handle:dword ); @returns( "eax" );`

This function positions the file pointer of the file specified by the handle parameter to the end of that file. The file should have been opened for writing.

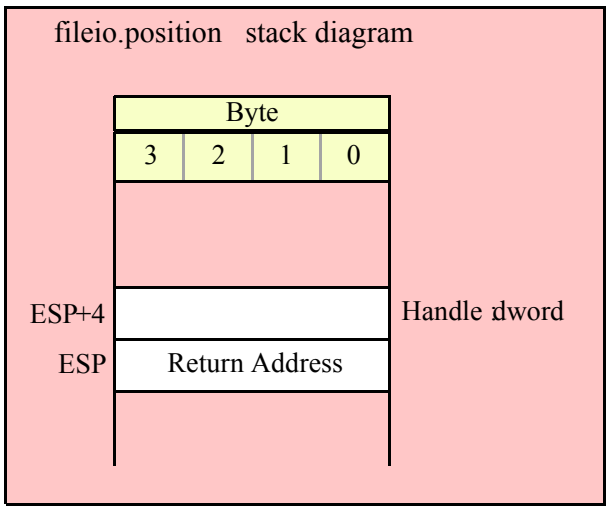
```
HLA high-level calling sequence examples:

fileio.append( fileHandle );

// If the file handle is in a register (EAX):

fileio.append( eax );


HLA low-level calling sequence examples:


push( fileHandle );
call fileio.append;

// If the file handle is in a register (EAX):

push( eax );
call fileio.append;
```



`fileio.position( Handle:dword ); @returns( "eax" );`

This function returns the file position (in bytes) of the file specified by the handle parameter. It returns the file position offset in the EAX register.

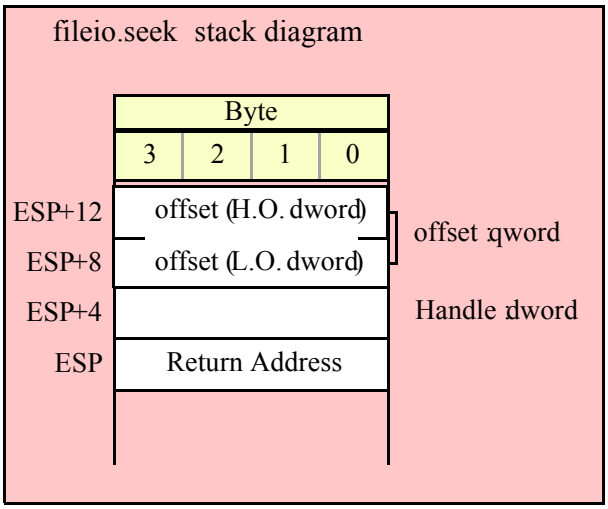
```
HLA high-level calling sequence examples:

fileio.position( fileHandle );
mov( eax, (type dword filePosition));

// If the file handle is in a register (EAX):

fileio.position( eax );
mov( eax, (type dword filePosition));


HLA low-level calling sequence examples:


push( fileHandle );
call fileio.position;
mov( eax, (type dword filePosition));

// If the file handle is in a register (EAX):

push( eax );
call fileio.position;
mov( eax, (type dword filePosition));
```



fileio.seek(Handle:dword; offset:qword); @returns("eax");

This function sets the file position in the file specified by the Handle parameter to the position specified by the offset parameter. The offset parameter specifies the file position in bytes from the beginning of the file. It returns the error status in EAX.

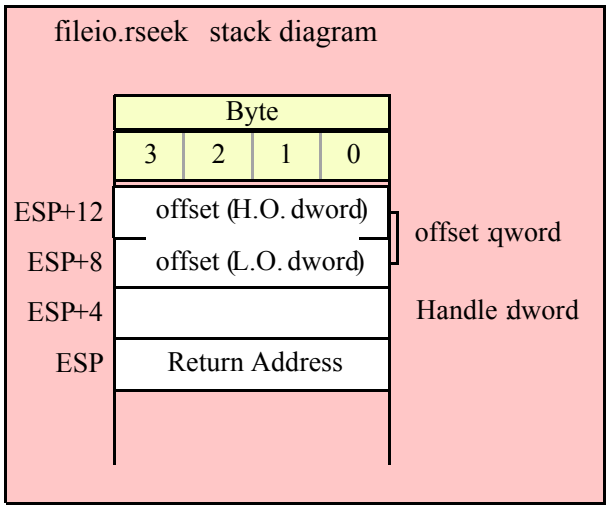
```
HLA high-level calling sequence examples:

fileio.seek( fileHandle, qwordOffsetVar );


HLA low-level calling sequence examples:

push( fileHandle );
push( (type dword qwordOffsetVar[4]));
push( (type dword qwordOffsetVar));
call fileio.seek;

// If the file handle is in a register (EAX):

push( eax );
push( (type dword qwordOffsetVar[4]));
push( (type dword qwordOffsetVar));
call fileio.seek;

// If the offset is in a register pair (EDX:EAX):

push( fileHandle );
push( edx );    // H.O. dword of offset
push( eax );    // L.O. dword of offset
call fileio.seek;
```



`fileio.rSeek( Handle:dword; offset:qword ); @returns( "eax" );`

This function sets the file position in the file specified by the Handle parameter to the position specified by the offset parameter. The offset parameter specifies the file position in bytes from the end of the file. It returns the error status in EAX.

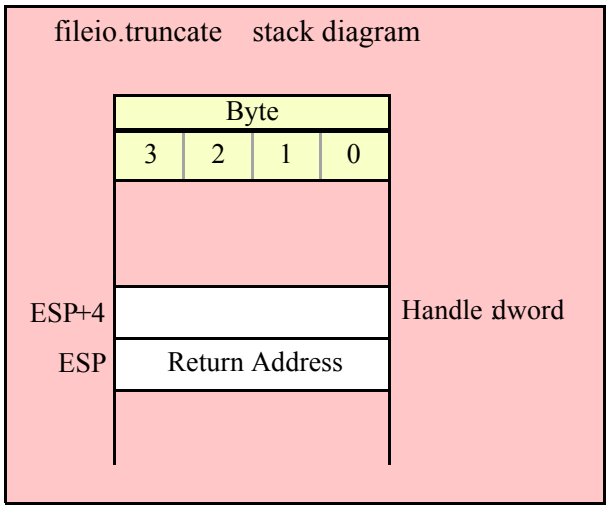
```
HLA high-level calling sequence examples:

fileio.rSeek( fileHandle, qwordOffsetVar );


HLA low-level calling sequence examples:

push( fileHandle );
push( (type dword qwordOffsetVar[4]));
push( (type dword qwordOffsetVar));
call fileio.rSeek;

// If the file handle is in a register (EAX):

push( eax );
push( (type dword qwordOffsetVar[4]));
push( (type dword qwordOffsetVar));
call fileio.rSeek;

// If the offset is in a register pair (EDX:EAX):

push( fileHandle );
push( edx );    // H.O. dword of offset
push( eax );    // L.O. dword of offset
call fileio.rSeek;
```



`fileio.truncate( Handle:dword ); @returns( "eax" );`

This function deletes all bytes in the file specified by the Handle parameter from the current file position to the end of the file. It returns the error status in EAX.

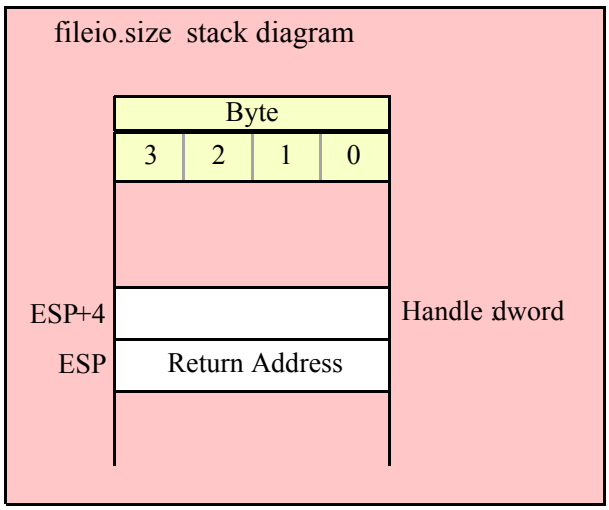
```
HLA high-level calling sequence examples:

fileio.truncate( fileHandle );

// If the file handle is in a register (EAX):

fileio.truncate( eax );


HLA low-level calling sequence examples:


push( fileHandle );
call fileio.truncate;

// If the file handle is in a register (EAX):

push( eax );
call fileio.truncate;
```



`fileio.size( Handle:dword ); @returns( "eax" );`

This function returns the current size of an open file whose handle you pass as a parameter. It returns the size in the EAX register. Note the overloaded version below.

```
HLA high-level calling sequence examples:

fileio.size( fileHandle );
mov( eax, fileSize );

// If the file handle is in a register (EAX):

fileio.size( eax );
mov( eax, fileSize );


HLA low-level calling sequence examples:


push( fileHandle );
call fileio.size;
mov( eax, fileSize );

// If the file handle is in a register (EAX):

push( eax );
call fileio.size;
mov( eax, fileSize );
```



15.3 File Output Routines

The file output routines in the fileio module are very similar to the file output routines in the file class module as well as the output routines in the standard output library module. In general, these routines require (at least) two parameters; the first is the file handle that you obtain via the fileio.open or fileio.openNew call, the second parameter is usually the value to write to the file. Some function contain additional parameters that provide formatting information. Note that these functions require that you've opened the file for writing, reading and writing, for for appending. If the file is not open or you've only opened it for reading, these routines will raise an appropriate exception.

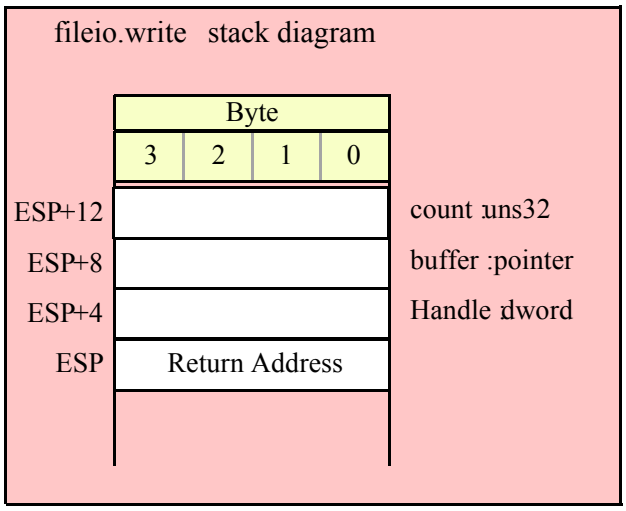
15.3.1 Miscellaneous Output Routines

```
fileio.write( Handle:dword; var buffer:var; count:uns32 );
```

This procedure writes the number of bytes specified by the count variable to the file. The bytes starting at the address of the buffer byte are written to the file. No range checking is done on the buffer, it is your responsibility to ensure that the buffer contains at least count valid data bytes. Note that buffer is an untyped reference parameter. This means that fileio.write will take the address of whatever object you provide as this parameter (including pointer variables, which may not be what you want). If you want to pass the value of a pointer variable as the buffer address (rather than the address of the pointer variable) when using the high-level style calling syntax, use the VAL keyword as a prefix to the parameter (see the following examples).

```
HLA high-level calling sequence examples:

fileio.write( fileHandle, buffer, count );

// If bufPtr is a dword object containing the
// address of the buffer whose data you wish to
// write to the file:

fileio.write( fileHandle, val bufPtr, count );

// The following writes the four-byte value of
// the bufPtr variable to the file (an unusual
// operation):

fileio.write( fileHandle, bufPtr, 4 );


HLA low-level calling sequence examples:
```

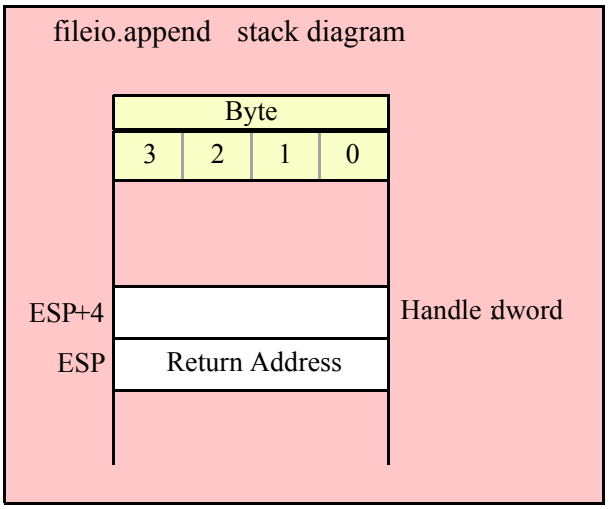
```
// Assumes buffer is a static object at a fixed
// address in memory:

push( fileHandle );
pushd( &buffer );
push( count );
call fileio.write;

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

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

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

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

// If bufPtr points at the buffer to write,
// then use code like this:

push( fileHandle );
push( bufPtr);
push( count );
call fileio.write;

  // To write the 4 bytes at bufPtr to
  // the file (unusual), you could use
  // code like this:

push( fileHandle );
lea( eax, bufPtr );
push( eax );
pushd( 4 );
call fileio.write;
```



`fileio.newln( Handle:dword )`

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

```
HLA high-level calling sequence examples:

fileio.newln( fileHandle );

// If the file handle is in a register (EAX):

fileio.newln( eax );


HLA low-level calling sequence examples:


push( fileHandle );
call fileio.newln;

// If the file handle is in a register (EAX):

push( eax );
call fileio.newln;
```

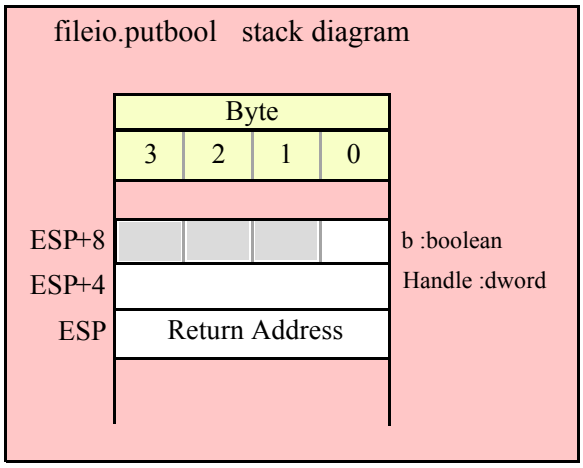


fileio.putbool(Handle:dword; b:boolean)

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

```
HLA high-level calling sequence examples:

fileio.putbool( fileHandle, boolVar );

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

fileio.putbool( fileHandle, al );


HLA low-level calling sequence examples:


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

push( fileHandle );
push( (type dword boolVar ) );
call fileio.putbool;

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

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

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

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

```
movzx( boolVar , eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putbool;

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

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

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

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



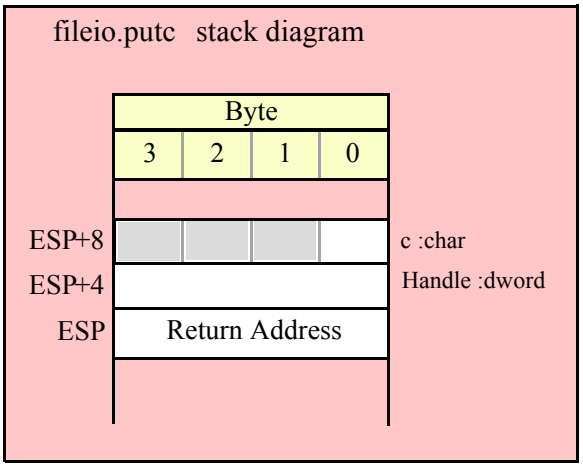
15.3.2 Character, String, and Character Set Output Routines

`fileio.putc( Handle:dword; c:char )`

Writes the character specified by the c parameter to the file specified by the Handle parameter.

```
HLA high-level calling sequence examples:

fileio.putc( fileHandle, charVar );

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

fileio.putc( fileHandle, al );


HLA low-level calling sequence examples:
```

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

push( fileHandle );
push( (type dword charVar) );
call fileio.putc;

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

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

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

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

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

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

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

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



fileio.putcSize(Handle:dword; c:char; width:int32; fill:char)

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

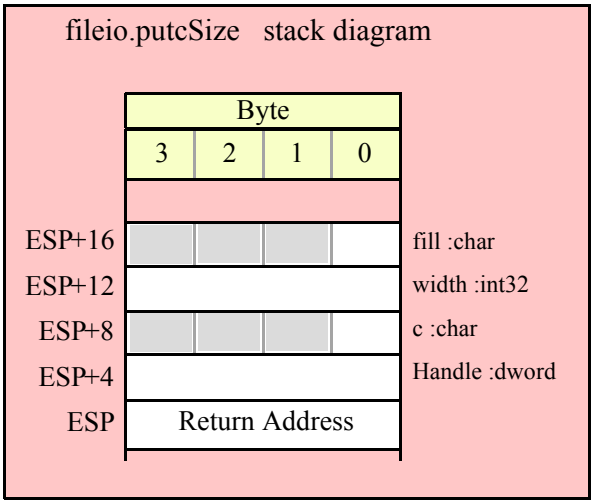
HLA high-level calling sequence examples:

```
fileio.putcSize( fileHandle, charVar, width, padChar );
```

HLA low-level calling sequence examples:

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

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

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

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

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

push( fileHandle );
sub( 12, esp );
push( eax );
movzx( charVar, eax );
mov( eax, [esp+12] );
mov( width, eax );
mov( eax, [esp+8] );
movzx( padChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putcSize;

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

push( fileHandle );
push( eax );   // Assume charVar is in AL
```

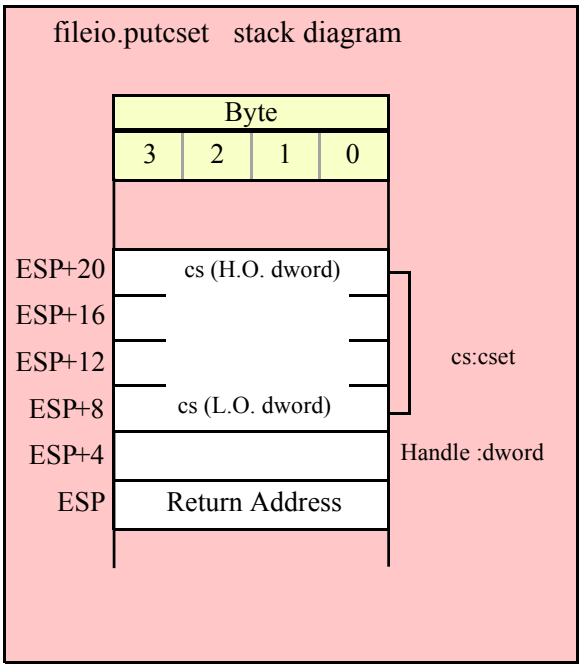
```
push( width );
push( ebx );   // Assume padChar is in BL
call fileio.putcSize;

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

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



fileio.putcset(Handle:dword; cst:cset)

This function writes all the members of the cst character set parameter to the file specified by the Handle variable.

```
HLA high-level calling sequence examples:

fileio.putcset( fileHandle, csVar );
fileio.putcset( fileHandle, [ebx] ); // EBX points at the cset.


HLA low-level calling sequence examples:

push( fileHandle );
push( (type dword csVar[12]) );  // Push H.O. dword first
push( (type dword csVar[8]) );
push( (type dword csVar[4]) );
push( (type dword csVar) );      // Push L.O. dword last
call fileio.putcset;
```

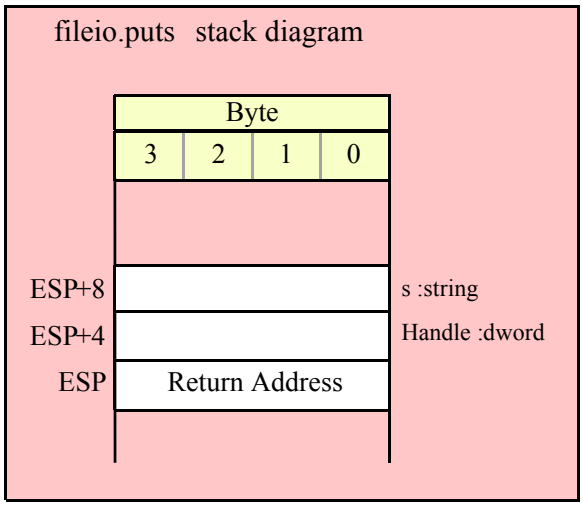


`fileio.puts( Handle:dword; s:string  )`

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

```
HLA high-level calling sequence examples:

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


HLA low-level calling sequence examples:

// For string variables:

push( fileHandle );
push( strVar );
call fileio.puts;

// For string values held in registers:

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

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

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

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

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

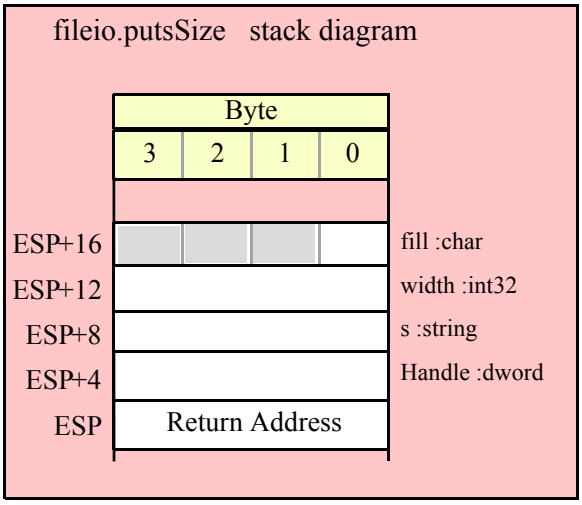


fileio.putsSize(Handle:dword; s:string; width:int32; fill:char)

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

```
HLA high-level calling sequence examples:

fileio.putsSize( fileHandle, strVar, width, ' ' );

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

fileio.putsSize( fileHandle, ebx, ecx, al );

fileio.putsSize( fileHandle, "Hello World", 25, padChar );


HLA low-level calling sequence examples:

// For string variables:

push( fileHandle );
push( strVar );
```

```
push( width );
pushd( ' ' );
call fileio.putsSize;

// For string values held in registers:

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

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

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

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

readonly
  literalString :string := "Hello World";

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



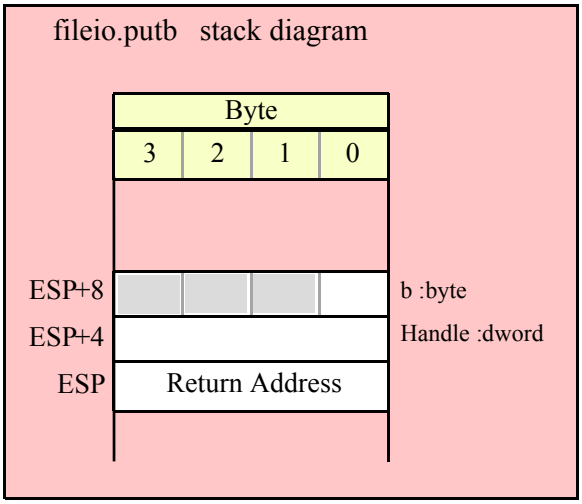
15.3.3 Hexadecimal Output Routines

fileio.putb(Handle:dword; b:byte)

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

```
HLA high-level calling sequence examples:


fileio.putb( fileHandle, byteVar );

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

fileio.putb( fileHandle, al );



HLA low-level calling sequence examples:


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

push( fileHandle );
push( (type dword byteVar ) );
call fileio.putb;

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

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

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

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

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

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

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

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

```
xchg( al, ah ); // Restore al/ah
call fileio.putb;
```

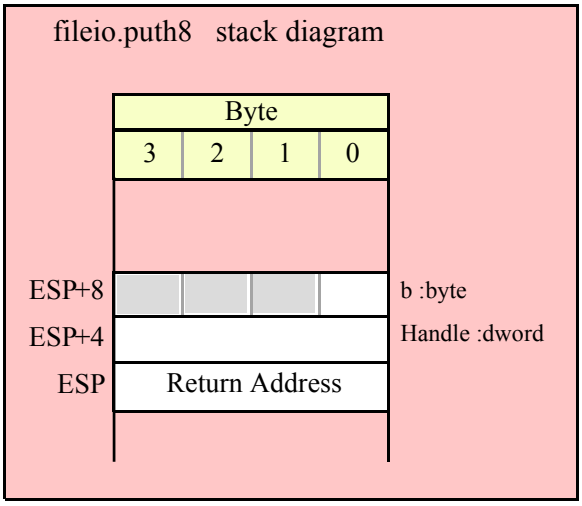


`fileio.puth8( Handle:dword; b:byte  )`

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

```
HLA high-level calling sequence examples:


fileio.puth8( fileHandle, byteVar );

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

fileio.puth8( fileHandle, al );


HLA low-level calling sequence examples:


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

push( fileHandle );
push( (type dword byteVar ) );
call fileio.puth8;

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

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

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

push( fileHandle );
```

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

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

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

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

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

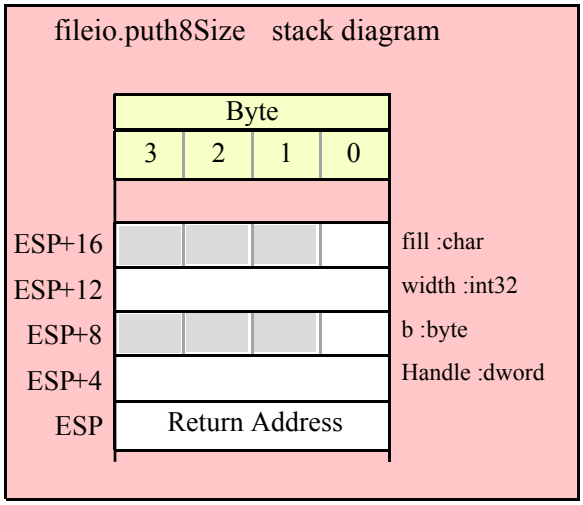


fileio.puth8Size(Handle:dword; b:byte; size:dword; fill:char)

The fileio.puth8Size function writes an 8-bit hexadecimal value to a file allowing you specify a minimum field width and a fill character.

```
HLA high-level calling sequence examples:

fileio.puth8Size( fileHandle, byteVar, width, padChar );

HLA low-level calling sequence examples:


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

```
push( fileHandle );
push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call fileio.puth8Size;

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

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

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

push( fileHandle );
sub( 12, esp );
push( eax );
movzx( byteVar, eax );
mov( eax, [esp+12] );
mov( width, eax );
mov( eax, [esp+8] );
movzx( padChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puth8Size;

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

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

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

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

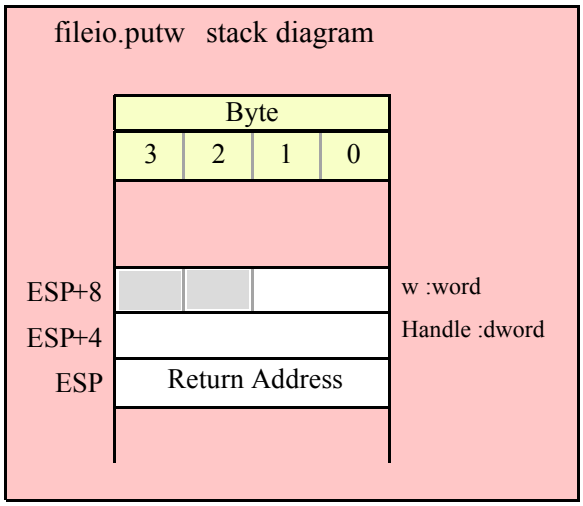


fileio.putw(Handle:dword; w:word)

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

```
HLA high-level calling sequence examples:


fileio.putw( fileHandle, wordVar );

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

fileio.putw( fileHandle, ax );



HLA low-level calling sequence examples:


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

push( fileHandle );
push( (type dword wordVar) );
call fileio.putw;

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

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

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

push( fileHandle );
```

```
    sub( 4, esp );
    push( eax );
    movzx( wordVar, eax );
    mov( eax, [esp+4] );
    pop( eax );
    call fileio.putw;

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

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

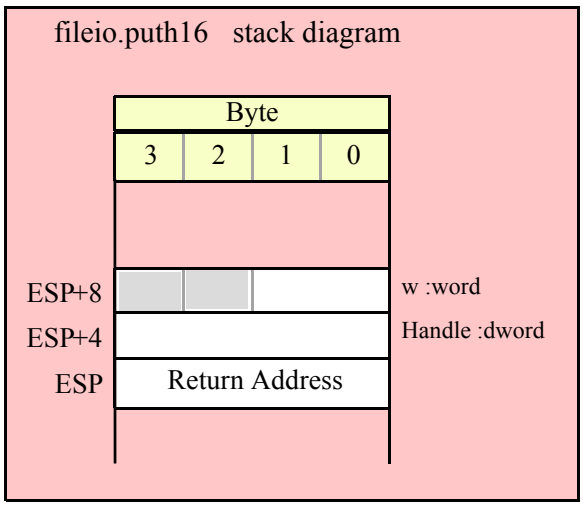


fileio.puth16(Handle:dword; w:word)

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

```
    HLA high-level calling sequence examples:


    fileio.puth16( fileHandle, wordVar );

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

    fileio.puth16( fileHandle, ax );


    HLA low-level calling sequence examples:


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

    push( fileHandle );
    push( (type dword wordVar) );
    call fileio.puth16;

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

```
// the following:

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

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

push( fileHandle );
sub( 4, esp );
push( eax );
movzx( wordVar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puth16;

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

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

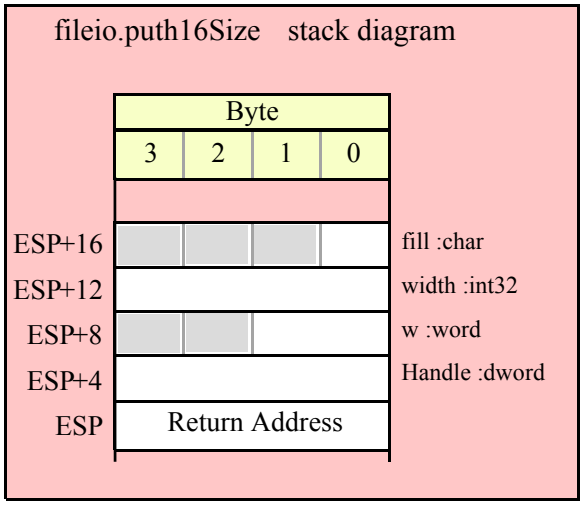


fileio.puth16Size(Handle:dword; w:word; size:dword; fill:char)

The fileio.puth16Size function writes a 16-bit hexadecimal value to a file allowing you specify a minimum field width and a fill character.

```
HLA high-level calling sequence examples:

fileio.puth16Size( fileHandle, wordVar, width, padChar );


HLA low-level calling sequence examples:


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

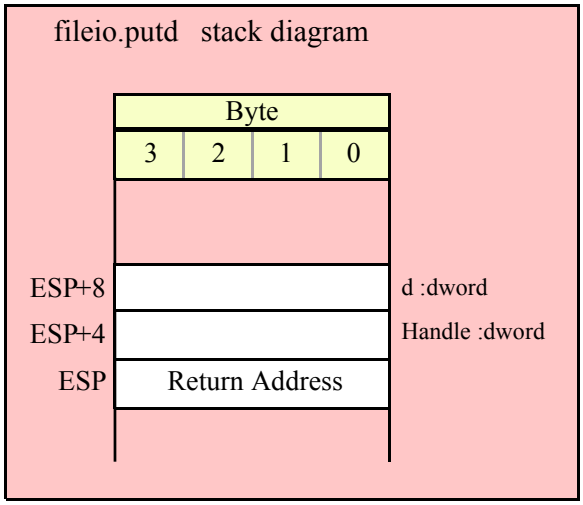
```
push( fileHandle );
push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call fileio.puth16Size;

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

push( fileHandle );
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.puth16Size;

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

push( fileHandle );
sub( 12, esp );
push( eax );
movzx( wordVar, eax );
mov( eax, [esp+12] );
mov( width, eax );
mov( eax, [esp+8] );
movzx( padChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puth16Size;

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

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



`fileio.putd( Handle:dword; d:dword )`

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

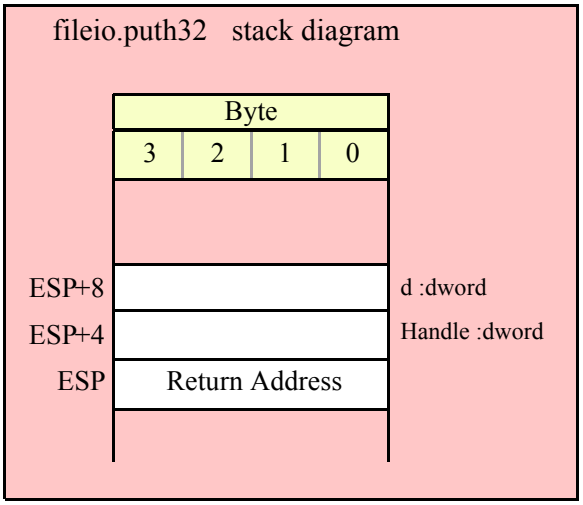
```
HLA high-level calling sequence examples:


fileio.putd( fileHandle, dwordVar );

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

fileio.putd( fileHandle, eax );



HLA low-level calling sequence examples:

push( fileHandle );
push( dwordVar );
call fileio.putd;

push( fileHandle );
push( eax );
call fileio.putd;
```



fileio.puth32(Handle:dword; d:dword)

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

```
HLA high-level calling sequence examples:


fileio.puth32( fileHandle, dwordVar );

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

fileio.puth32( fileHandle, eax );



HLA low-level calling sequence examples:

push( fileHandle );
push( dwordVar );
call fileio.puth32;

push( fileHandle );
push( eax );
call fileio.puth32;
```

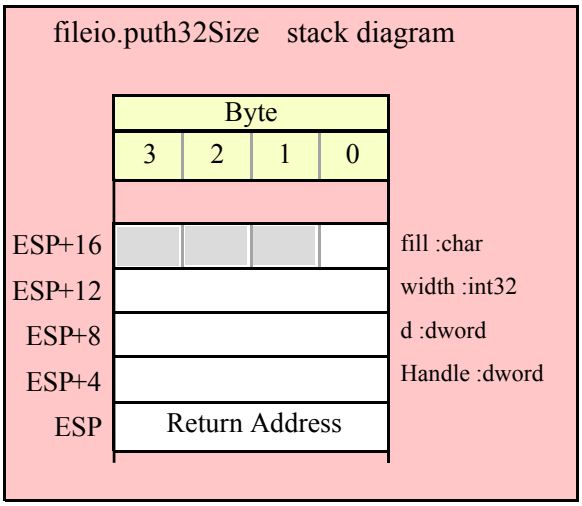


fileio.puth32Size(Handle:dword; d:dword; size:dword; fill:char)

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

```
HLA high-level calling sequence examples:


fileio.puth32Size( fileHandle, dwordVar, width, ' ' );

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

fileio.puth32Size( fileHandle, eax, width, cl );



HLA low-level calling sequence examples:

push( fileHandle );
push( dwordVar );
push( width );
pushd( ' ' );
call fileio.puth32Size;

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

// Assume fill char is in CH

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

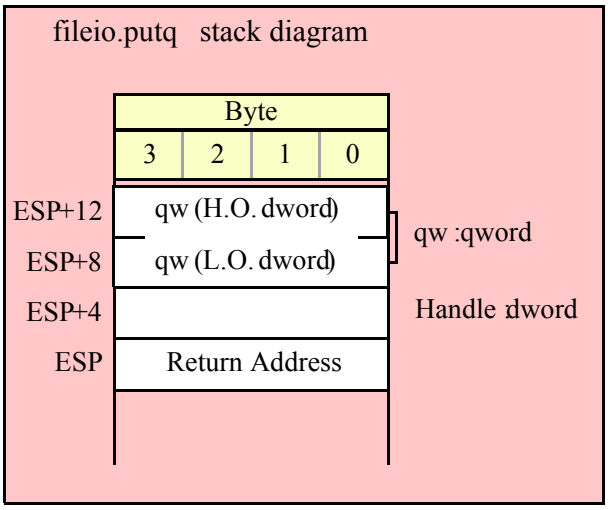
```
// Alternate method of the above

push( fileHandle );
push( eax );
push( width );
sub( 4, esp );
mov( ch, [esp] );
call fileio.puth32Size;

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

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


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

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

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

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



fileio.putq(Handle:dword; q:qword)

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

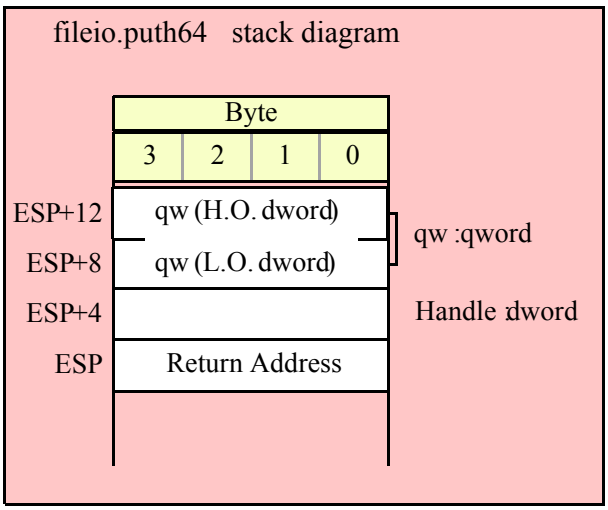
```
HLA high-level calling sequence examples:


fileio.putq( fileHandle, qwordVar );


HLA low-level calling sequence examples:

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



fileio.puth64(Handle:dword; q:qword)

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

```
HLA high-level calling sequence examples:


fileio.puth64( fileHandle, qwordVar );


HLA low-level calling sequence examples:

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

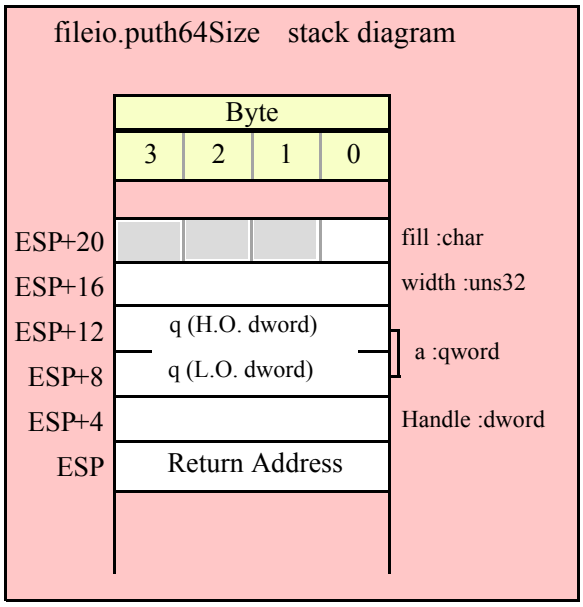


fileio.puth64Size(Handle:dword; q:qword; size:dword; fill:char)

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

```
HLA high-level calling sequence examples:


fileio.puth64Size( fileHandle, qwordVar, width, ' ' );



HLA low-level calling sequence examples:

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



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

// Assume fill char is in CH

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

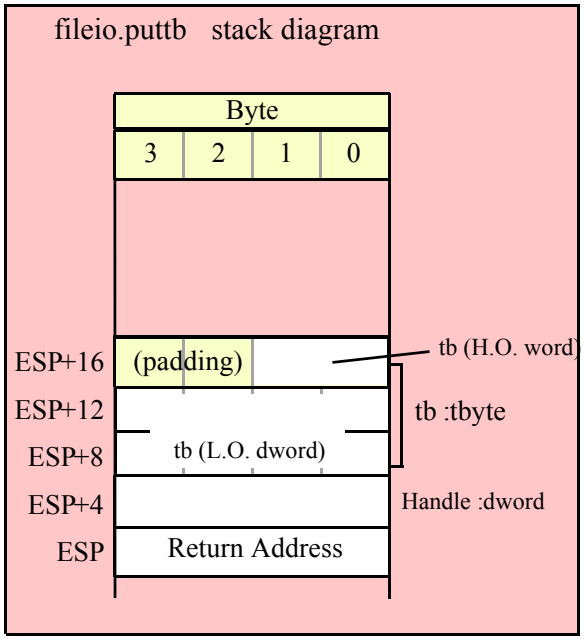
```
// Alternate method of the above

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

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

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


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

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

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

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



`fileio.puttb( Handle:dword; tb:tbyte )`

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

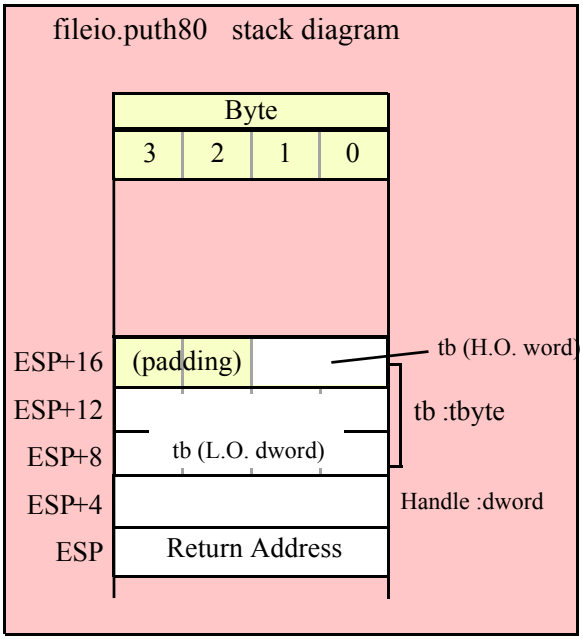
```
HLA high-level calling sequence examples:


fileio.puttb( fileHandle, tbyteVar );



HLA low-level calling sequence examples:

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



`fileio.puth80( Handle:dword; tb:tbyte )`

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

```
HLA high-level calling sequence examples:


fileio.puth80( fileHandle, tbyteVar );



HLA low-level calling sequence examples:

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

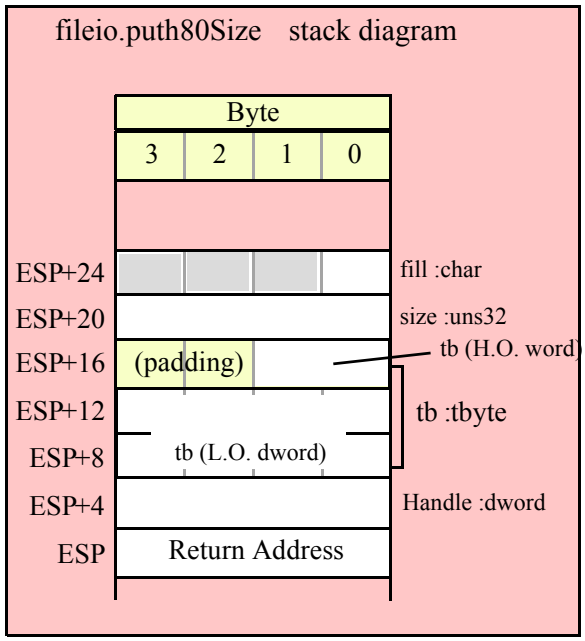


`fileio.puth80Size( Handle:dword; tb:tbyte; size:dword; fill:char )`

The fileio.puth80Size function lets you specify a minimum field width and a fill character. It writes the tbyte value tb as a hexadecimal string to the specified file using the provided minimum size and fill character.

```
HLA high-level calling sequence examples:


fileio.puth80Size( fileHandle, tbyteVar, width, ' ' );



HLA low-level calling sequence examples:

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



// Assume fill char is in CH

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

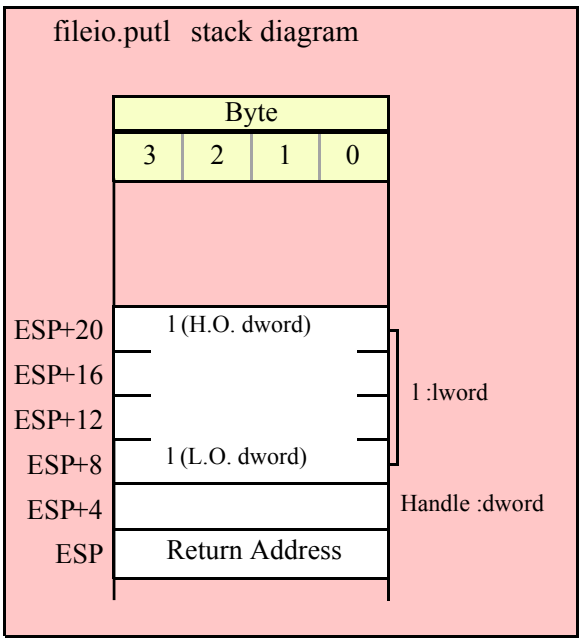
```
call fileio.puth80Size;

// Alternate method of the above

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

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

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


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

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

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

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



`fileio.putl( Handle:dword; l:lword )`

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

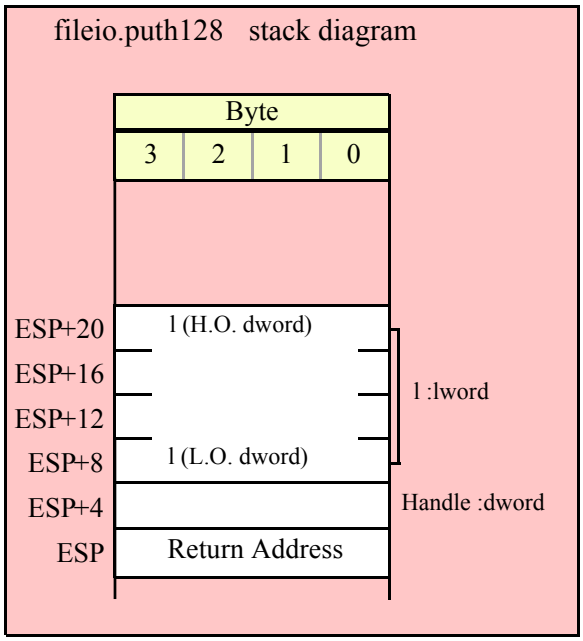
```
HLA high-level calling sequence examples:


fileio.putl( fileHandle, lwordVar );



HLA low-level calling sequence examples:

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



`fileio.puth128( Handle:dword; l:lword )`

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

```
HLA high-level calling sequence examples:


fileio.puth128( fileHandle, lwordVar );



HLA low-level calling sequence examples:

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

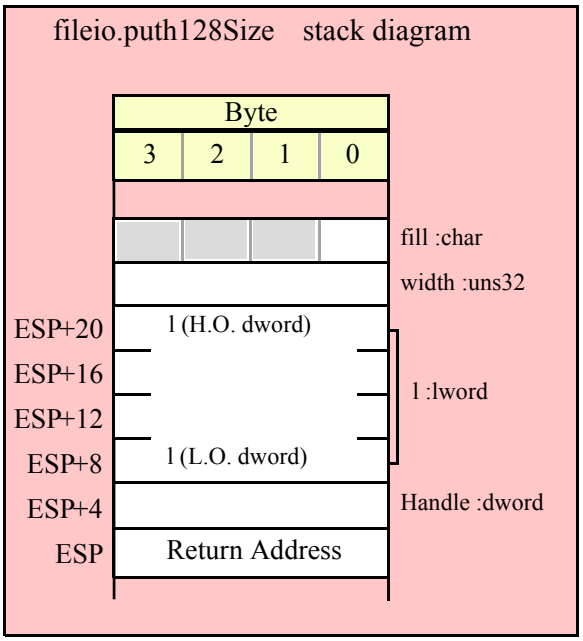


`fileio.puth128Size( Handle:dword; l:lword; size:dword; fill:char )`

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

```
HLA high-level calling sequence examples:


fileio.puth128Size( fileHandle, tbyteVar, width, ' ' );



HLA low-level calling sequence examples:

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



// Assume fill char is in CH

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

```
call fileio.puth128Size;

// Alternate method of the above

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

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

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


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

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

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

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



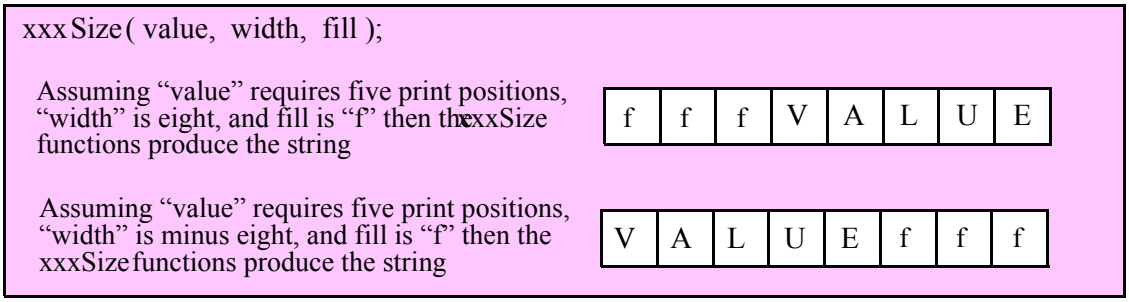
15.3.4 Signed Integer Output Routines

These routines convert signed integer values to string format and write that string to the file specified by the Handle parameter. The fileio.putxxxSize functions contain width and fill parameters that let you specify the minimum field width when outputting a value.

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

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

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



`fileio.puti8 ( Handle:dword; b:byte )`

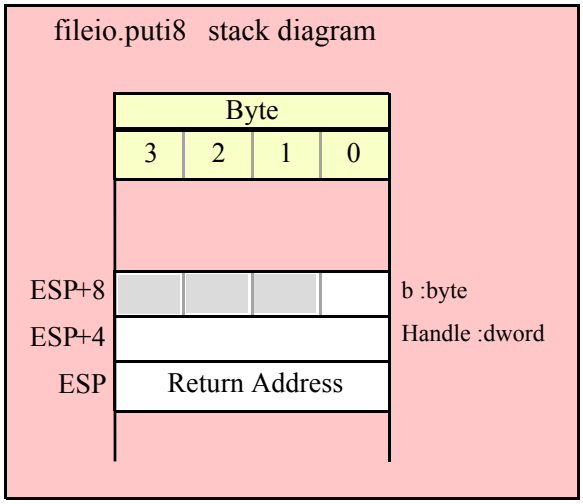
This function converts the eight-bit signed integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


fileio.puti8( fileHandle, byteVar );
```

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

fileio.puti8( fileHandle, al );
```

HLA low-level calling sequence examples:

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

push( fileHandle );
push( (type dword byteVar ) );
call fileio.puti8;

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

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

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

push( fileHandle );
sub( 4, esp );
push( eax );
movzx( byteVar , eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puti8;

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

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

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

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

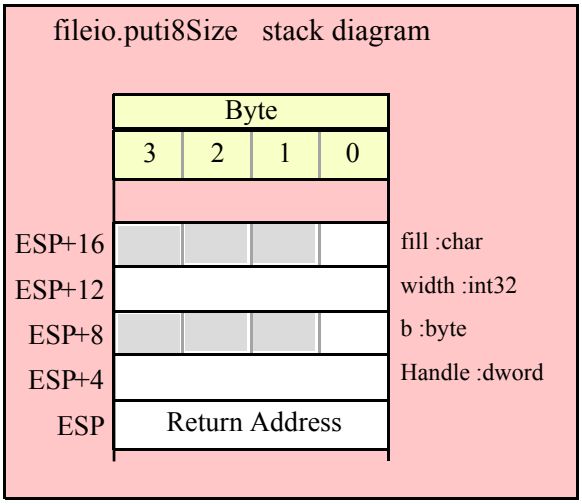


fileio.puti8Size (Handle:dword; b:byte; width:int32; fill:char)

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

```
HLA high-level calling sequence examples:

fileio.puti8Size( fileHandle, byteVar, width, padChar );


HLA low-level calling sequence examples:


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

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

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

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

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

push( fileHandle );
sub( 12, esp );
push( eax );
```

```
movzx( byteVar, eax );
mov( eax, [esp+12] );
mov( width, eax );
mov( eax, [esp+8] );
movzx( padChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puti8Size;

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

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

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

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



fileio.puti16(Handle:dword; w:word)

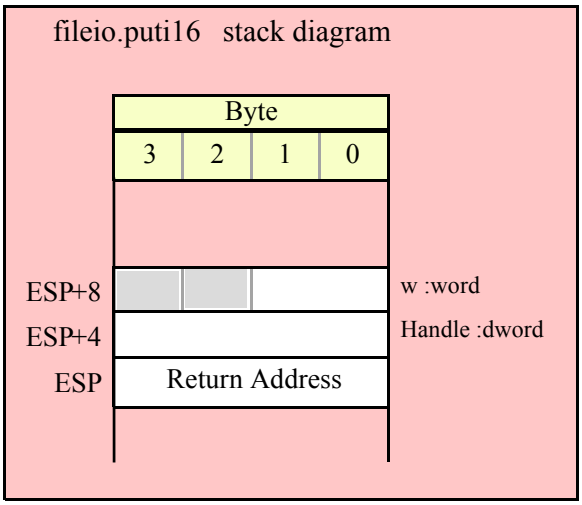
This function converts the 16-bit signed integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

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

```
fileio.puti16( fileHandle, wordVar );

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

fileio.puti16( fileHandle, ax );
```

HLA low-level calling sequence examples:

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

push( fileHandle );
push( (type dword wordVar) );
call fileio.puti16;

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

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

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

push( fileHandle );
sub( 4, esp );
push( eax );
movzx( wordVar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puti16;

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

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

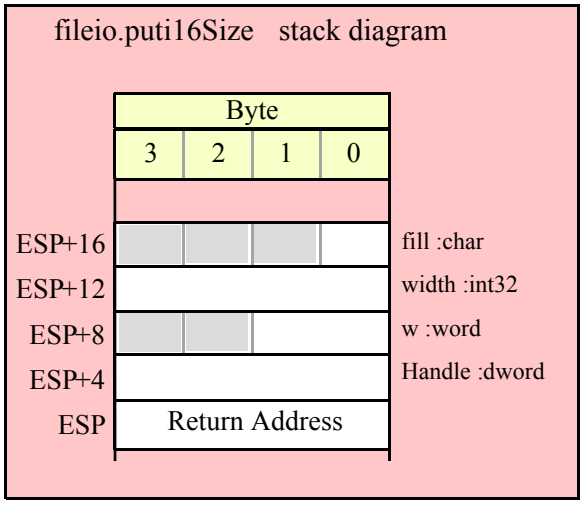


fileio.puti16Size(Handle:dword; w:word; width:int32; fill:char)

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

```
HLA high-level calling sequence examples:

fileio.puti16Size( fileHandle, wordVar, width, padChar );


HLA low-level calling sequence examples:


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

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

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

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

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

push( fileHandle );
```

```
    sub( 12, esp );
    push( eax );
    movzx( wordVar, eax );
    mov( eax, [esp+12] );
    mov( width, eax );
    mov( eax, [esp+8] );
    movzx( padChar, eax );
    mov( eax, [esp+4] );
    pop( eax );
    call fileio.puti16Size;

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

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

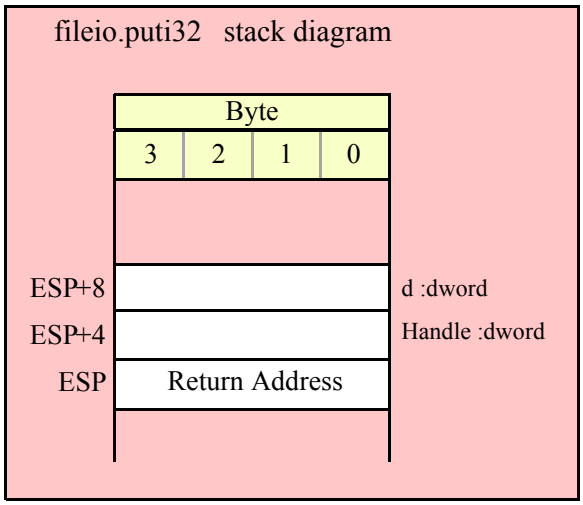


fileio.puti32(Handle:dword; d:dword)

This function converts the 32-bit signed integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

```
    HLA high-level calling sequence examples:


    fileio.puti32( fileHandle, dwordVar );

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

    fileio.puti32( fileHandle, eax );


    HLA low-level calling sequence examples:
```

```
push( fileHandle );
push( dwordVar );
call fileio.puti32;

push( fileHandle );
push( eax );
call fileio.puti32;
```

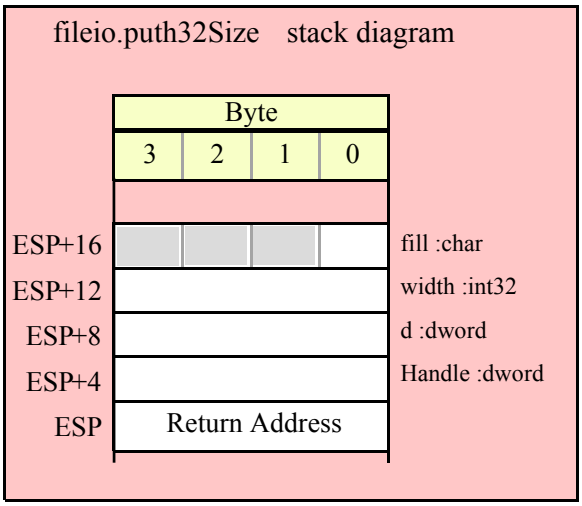


fileio.puti32Size(Handle:dword; d:dword; width:int32; fill:char)

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

```
HLA high-level calling sequence examples:


fileio.puti32Size( fileHandle, dwordVar, width, ' ' );

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

fileio.puti32Size( fileHandle, eax, width, cl );



HLA low-level calling sequence examples:

push( fileHandle );
push( dwordVar );
push( width );
pushd( ' ' );
call fileio.puti32Size;

push( fileHandle );
push( eax );
push( width );
push( ecx ); // fill char is in CL
call fileio.puti32Size;

// Assume fill char is in CH

push( fileHandle );
push( eax );
```

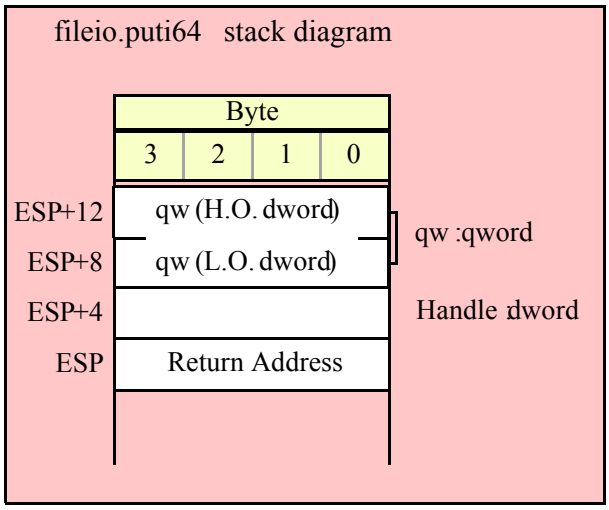
```
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call fileio.puti32Size;

// Alternate method of the above

push( fileHandle );
push( eax );
push( width );
sub( 4, esp );
mov( ch, [esp] );
call fileio.puti32Size;

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

push( fileHandle );
push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.puti32Size;


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

push( fileHandle );
push( eax );
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call fileio.puti32Size;

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

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



`fileio.puti64( Handle:dword; q:qword )`

This function converts the 64-bit signed integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


fileio.puti64( fileHandle, qwordVar );



HLA low-level calling sequence examples:

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

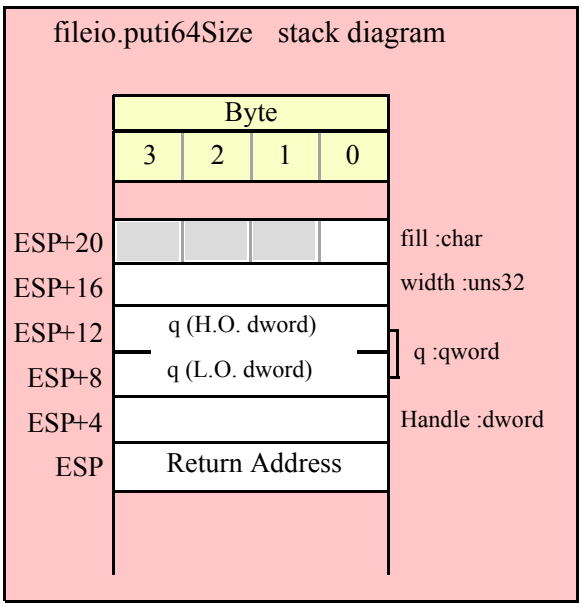


fileio.puti64Size(Handle:dword; q:qword; width:int32; fill:char)

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

```
HLA high-level calling sequence examples:


fileio.puti64Size( fileHandle, qwordVar, width, ' ' );



HLA low-level calling sequence examples:

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



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

// Assume fill char is in CH

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

// Alternate method of the above

push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call fileio.puti64Size;

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

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

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

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

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

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

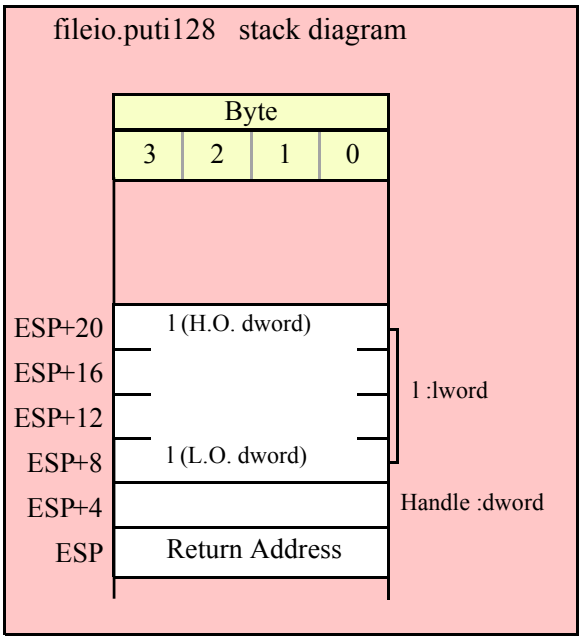


fileio.puti128(Handle:dword; l:lword)

This function converts the 128-bit signed integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


fileio.puti128( fileHandle, lwordVar );
```

```
HLA low-level calling sequence examples:

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



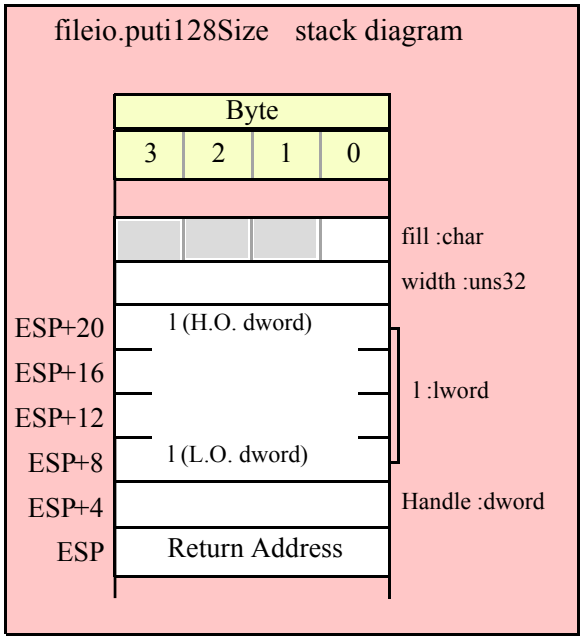
`fileio.puti128Size( Handle:dword; l:lword; width:int32; fill:char )`

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

```
HLA high-level calling sequence examples:


fileio.puti128Size( fileHandle, lwordVar, width, ' ' );



HLA low-level calling sequence examples:

push( fileHandle );
push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
push( width );
pushd( ' ' );
call fileio.puti128Size;



// Assume fill char is in CH

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

```
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call fileio.puti128Size;

// Alternate method of the above

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

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

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


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

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

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

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

```
pop( eax );
call fileio.puti128Size;
```

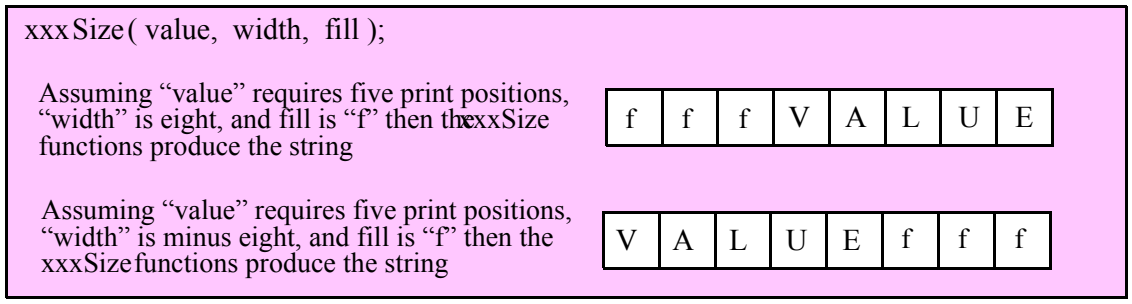


15.3.5 Unsigned Integer Output Routines

These routines convert unsigned integer values to string format and write that string to the file specified by the Handle parameter. The fileio.putxxxSize functions contain width and fill parameters that let you specify the minimum field width when outputting a value.

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

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

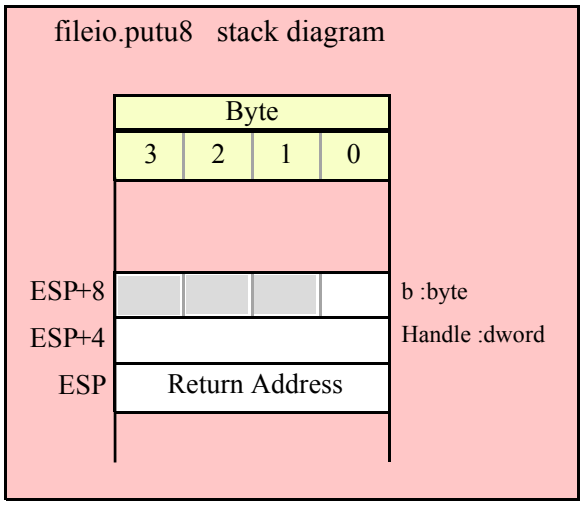


fileio.putu8 (Handle:dword; b:byte)

This function converts the eight-bit unsigned integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


fileio.putu8( fileHandle, byteVar );
```

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

fileio.putu8( fileHandle, al );



HLA low-level calling sequence examples:


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

push( fileHandle );
push( (type dword byteVar ) );
call fileio.putu8;

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

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

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

push( fileHandle );
sub( 4, esp );
push( eax );
movzx( byteVar , eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putu8;

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

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

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

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

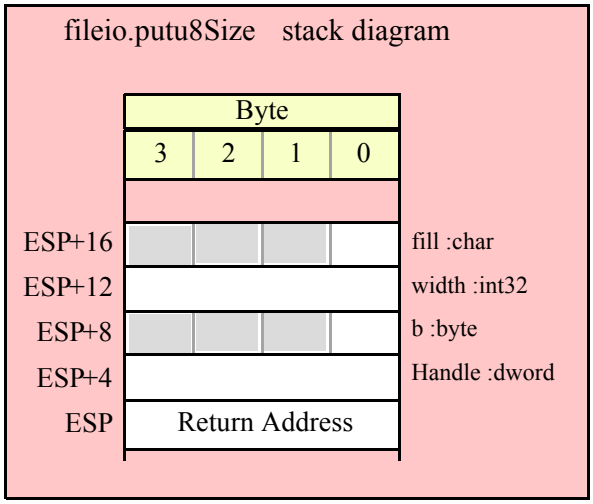


fileio.putu8Size(Handle:dword; b:byte; width:int32; fill:char)

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

```
HLA high-level calling sequence examples:

fileio.putu8Size( fileHandle, byteVar, width, padChar );


HLA low-level calling sequence examples:


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

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

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

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

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

push( fileHandle );
sub( 12, esp );
push( eax );
```

```
movzx( byteVar, eax );
mov( eax, [esp+12] );
mov( width, eax );
mov( eax, [esp+8] );
movzx( padChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putu8Size;

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

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

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

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



fileio.putu16(Handle:dword; w:word)

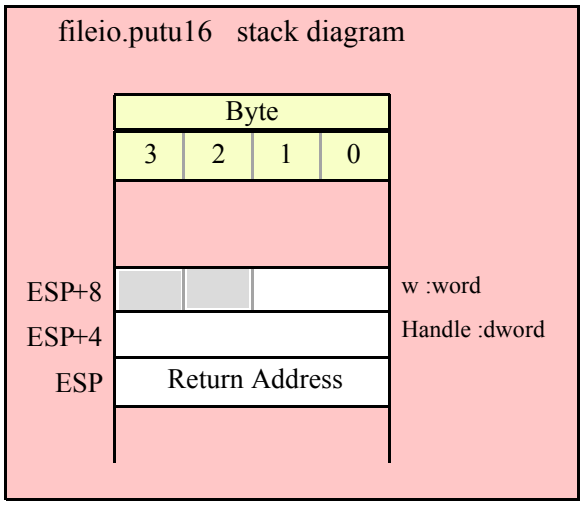
This function converts the 16-bit unsigned integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


fileio.putu16( fileHandle, wordVar );
```

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

fileio.putu16( fileHandle, ax );
```

HLA low-level calling sequence examples:

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

push( fileHandle );
push( (type dword wordVar) );
call fileio.putu16;

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

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

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

push( fileHandle );
sub( 4, esp );
push( eax );
movzx( wordVar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putu16;

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

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

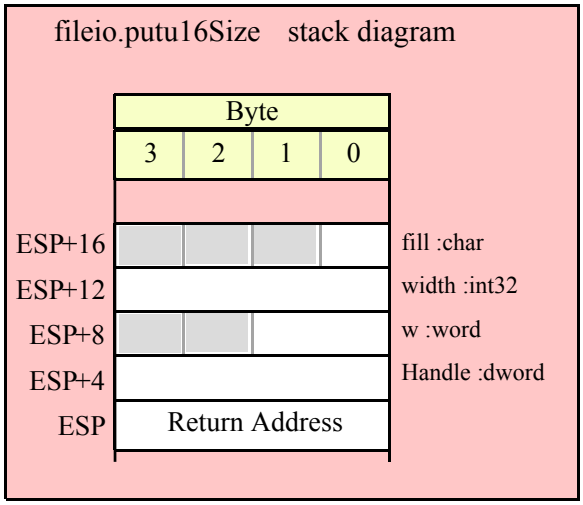


fileio.putu16Size(Handle:dword; w:word; width:int32; fill:char)

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

```
HLA high-level calling sequence examples:

fileio.putu16Size( fileHandle, wordVar, width, padChar );


HLA low-level calling sequence examples:


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

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

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

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

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

push( fileHandle );
sub( 12, esp );
```

```
push( eax );
movzx( wordVar, eax );
mov( eax, [esp+12] );
mov( width, eax );
mov( eax, [esp+8] );
movzx( padChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putu16Size;

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

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

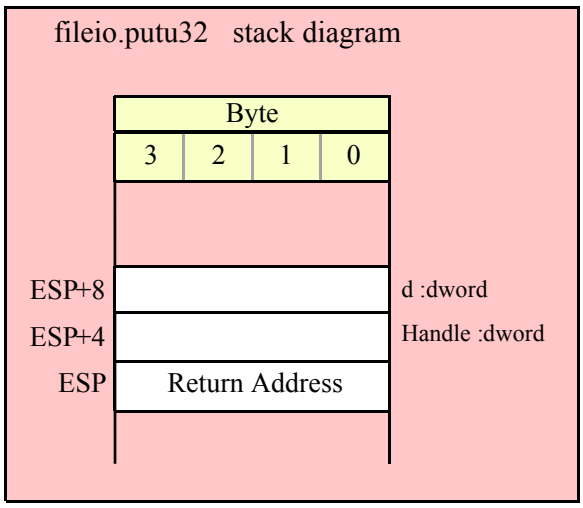


fileio.putu32(Handle:dword; d:dword)

This function converts the 32-bit unsigned integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


fileio.putu32( fileHandle, dwordVar );

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

fileio.putu32( fileHandle, eax );


HLA low-level calling sequence examples:

push( fileHandle );
```

```
push( dwordVar );
call fileio.putu32;

push( fileHandle );
push( eax );
call fileio.putu32;
```

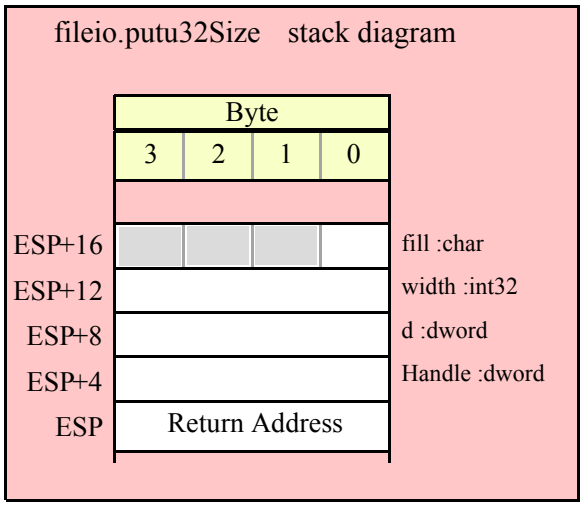


fileio.putu32Size(Handle:dword; d:dword; width:int32; fill:char)

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

```
HLA high-level calling sequence examples:


fileio.putu32Size( fileHandle, dwordVar, width, ' ' );

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

fileio.putu32Size( fileHandle, eax, width, cl );


HLA low-level calling sequence examples:

push( fileHandle );
push( dwordVar );
push( width );
pushd( ' ' );
call fileio.putu32Size;

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

// Assume fill char is in CH

push( fileHandle );
push( eax );
```

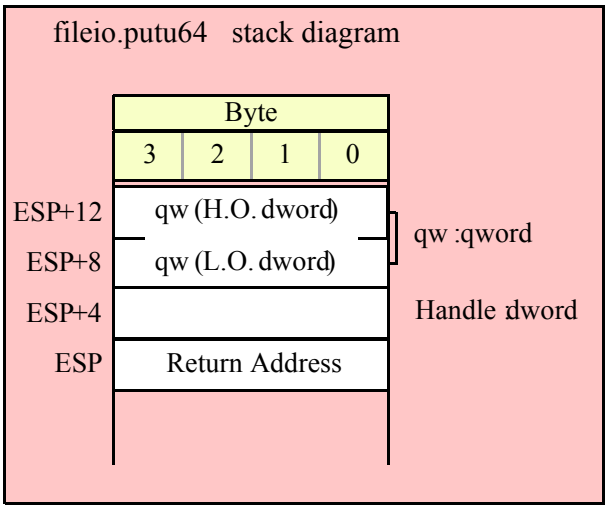
```
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call fileio.putu32Size;

// Alternate method of the above

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

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

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


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

push( fileHandle );
push( eax );
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call fileio.putu32Size;

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

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



`fileio.putu64( Handle:dword; q:qword )`

This function converts the 64-bit unsigned integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


fileio.putu64( fileHandle, qwordVar );



HLA low-level calling sequence examples:

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

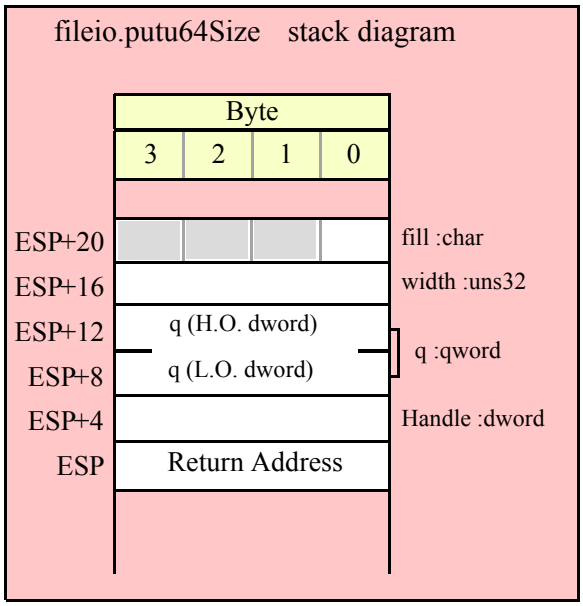


fileio.putu64Size(Handle:dword; q:qword; width:int32; fill:char)

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

```
HLA high-level calling sequence examples:


fileio.putu64Size( fileHandle, qwordVar, width, ' ' );



HLA low-level calling sequence examples:

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



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

// Assume fill char is in CH

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

// Alternate method of the above

push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call fileio.putu64Size;

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

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

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

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

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

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

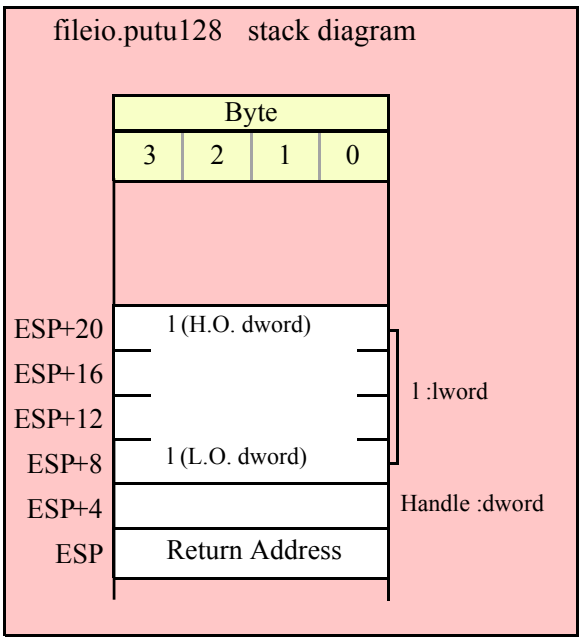


fileio.putu128(Handle:dword; l:lword)

This function converts the 128-bit unsigned integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


fileio.putu128( fileHandle, lwordVar );
```

```
HLA low-level calling sequence examples:

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



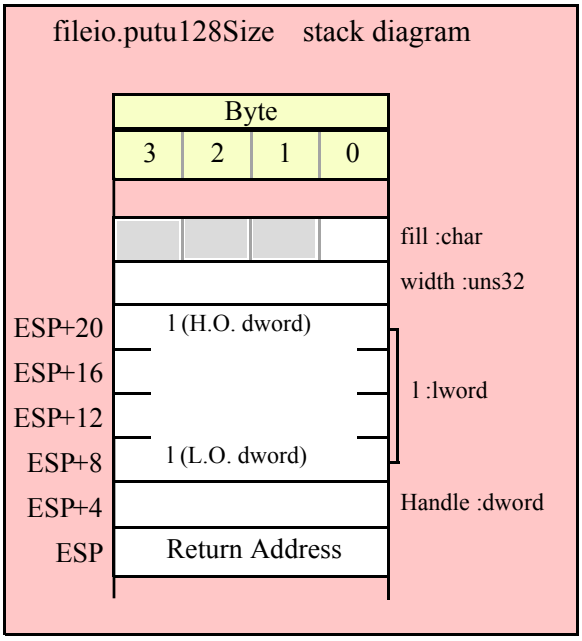
`fileio.putu128Size( Handle:dword; l:lword; width:int32; fill:char )`

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

```
HLA high-level calling sequence examples:


fileio.putu128Size( fileHandle, lwordVar, width, ' ' );



HLA low-level calling sequence examples:

push( fileHandle );
push( (type dword lwordVar[12]));  // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));     // L.O. dword last
push( width );
pushd( ' ' );
call fileio.putu128Size;


// Assume fill char is in CH

push( fileHandle );
```

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

// Alternate method of the above

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

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

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


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

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

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

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

```
mov( eax, [esp+4] );
pop( eax );
call fileio.putu128Size;
```



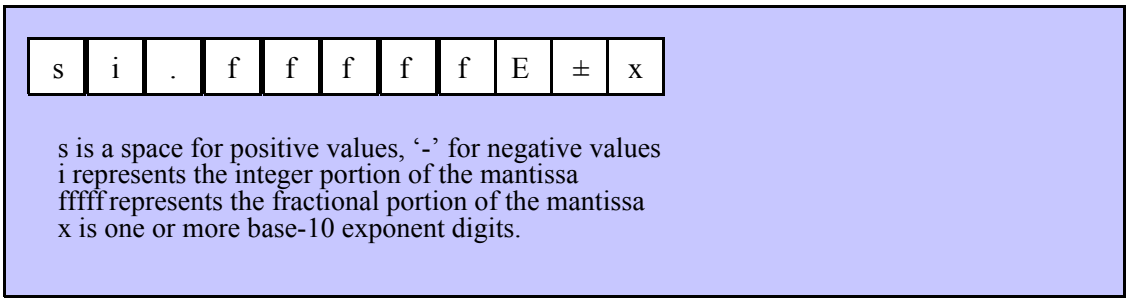
15.3.6 Floating Point Output Routines

The HLA file I/O class provides several procedures you can use to write floating point files to a text file. The following subsections describe these routines.

15.3.6.1 Real Output Using Scientific Notation

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

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

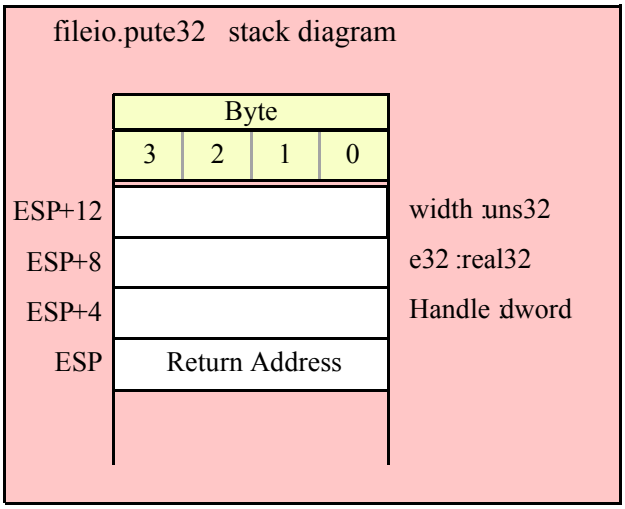


`fileio.pute32( Handle:dword; r:real32; width:uns32 )`

This function writes the 32-bit single precision floating point value passed in r to the file using scientific/exponential notation. This procedure prints the value using width print positions in the file. width should have a

minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 32-bit extended precision floating point values support about 6-7 significant digits. So a width value that yeilds more than seven mantissa digits will produce garbage output in the low order digits of the number.

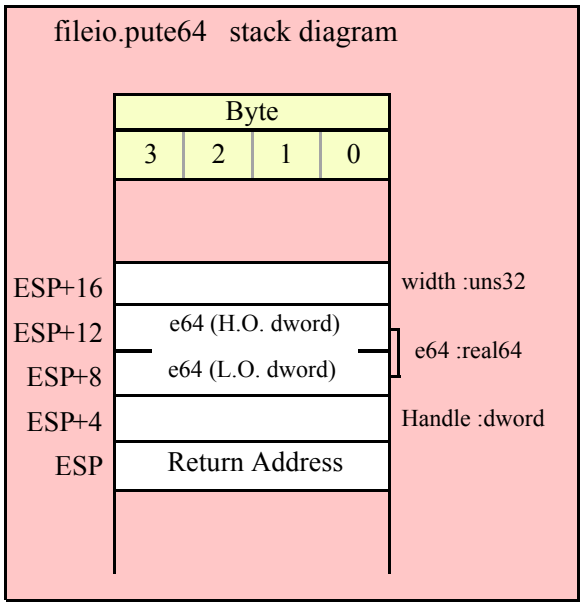
```
HLA high-level calling sequence examples:


fileio.pute32( fileHandle, r32Var, width );

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

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



HLA low-level calling sequence examples:

push( fileHandle );
push( (type dword r32Var) );
push( width );
call fileio.pute32;

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



fileio.pute64(Handle:dword; r:real64; width:uns32)

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

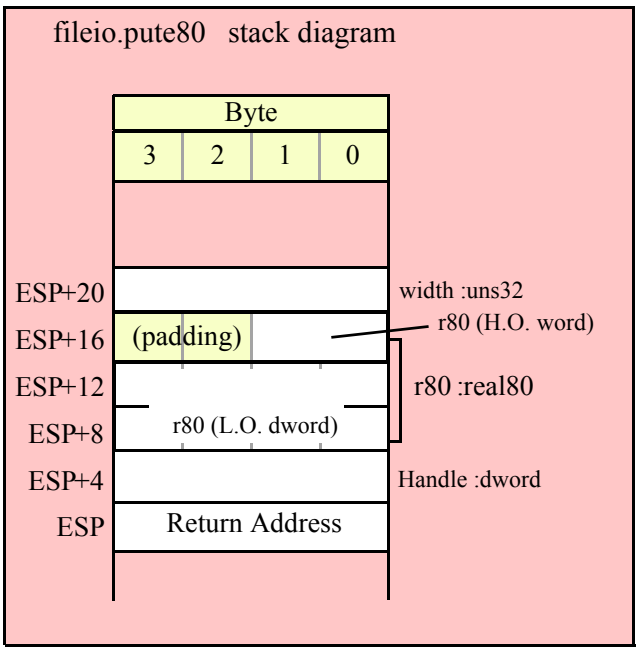
```
HLA high-level calling sequence examples:


fileio.pute64( fileHandle, r64Var, width );

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

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



HLA low-level calling sequence examples:

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

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



`fileio.pute80( Handle:dword; r:real80; width:uns32 )`

This function writes the 80-bit extended precision floating point value passed in r to the file using scientific/exponential notation. This procedure prints the value using width print positions in the file. width should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 80-bit extended precision floating point values support about 18 significant digits. So a width value that yeilds more than 18 mantissa digits will produce garbage output in the low order digits of the number.

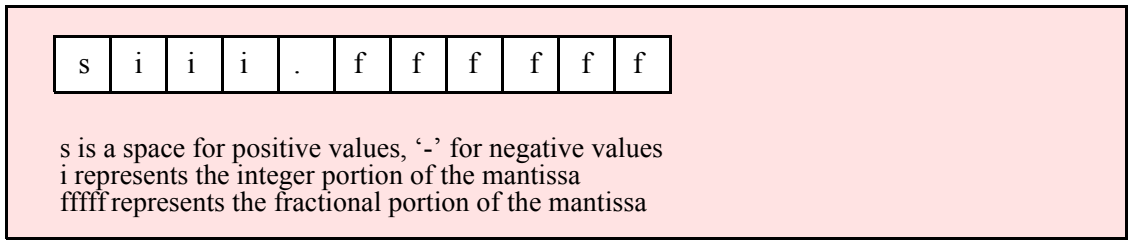
```
HLA high-level calling sequence examples:


fileio.pute80( fileHandle, r80Var, width );

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

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


HLA low-level calling sequence examples:

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

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



15.3.6.2 Real Output Using Decimal Notation

Although scientific (exponential) notation is the most general display format for real numbers, real numbers you display in this format are very difficult to read. Therefore, the HLA fileio module also provides a set of functions that output real values using the decimal representation. Although you cannot (practically) use these decimal output routines for all real values, they are applicable to a wide variety of common numbers you will use in your programs.

These functions come in two varieties. The first variety requires five 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. The second variety only requires the first four parameters and assumes the padding character is a space. These functions write their values using the following string format:

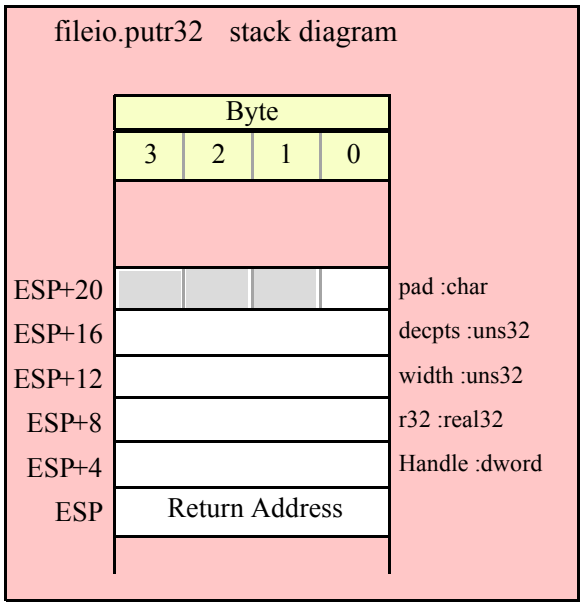


fileio.putr32(Handle:dword; r:real32; width:uns32; decpts:uns32; pad:char)

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

```
HLA high-level calling sequence examples:


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

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

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



HLA low-level calling sequence examples:

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

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

push( fileHandle );
sub( 4, esp );
fstp( (type real32 [esp]) );
pushd( 12 );
sub( 4, esp );
push( eax );
movzx( fill, eax ):// If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
call fileio.putr32;
```



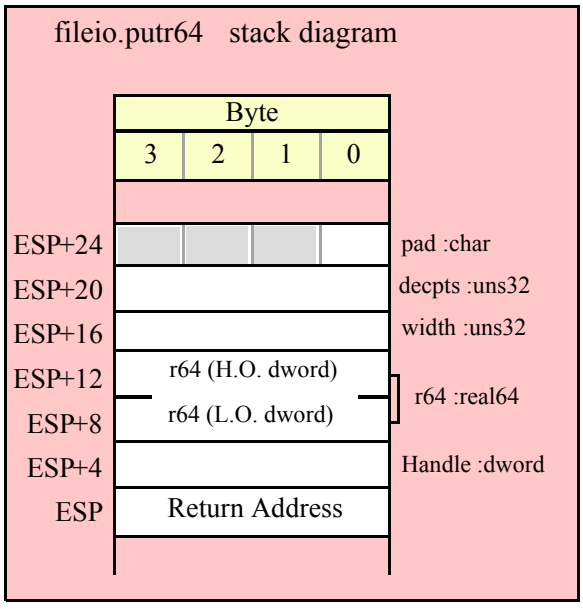
fileio.putr64(Handle:dword; r:real64; width:uns32; decpts:uns32; pad:char)

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

HLA high-level calling sequence examples:

```
fileio.putr64( fileHandle, r64Var, width, decpts, fill );
fileio.putr64( fileHandle, r64Var, 10, 2, '*' );

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

var
  r64Temp:real64;
  .
  .
```

```
  .
fstp( r64Temp );
fileio.putr64( fileHandle, r64Temp, 12, 2, al );


HLA low-level calling sequence examples:

push( fileHandle );
push( (type dword r64Var[4]) );
push( (type dword r64Var) );
push( width );
push( decpts );
movzx( fill, eax );
push( eax );
call fileio.putr64;

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

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



fileio.putr80(Handle:dword; r:real80; width:uns32; decpts:uns32; pad:char)

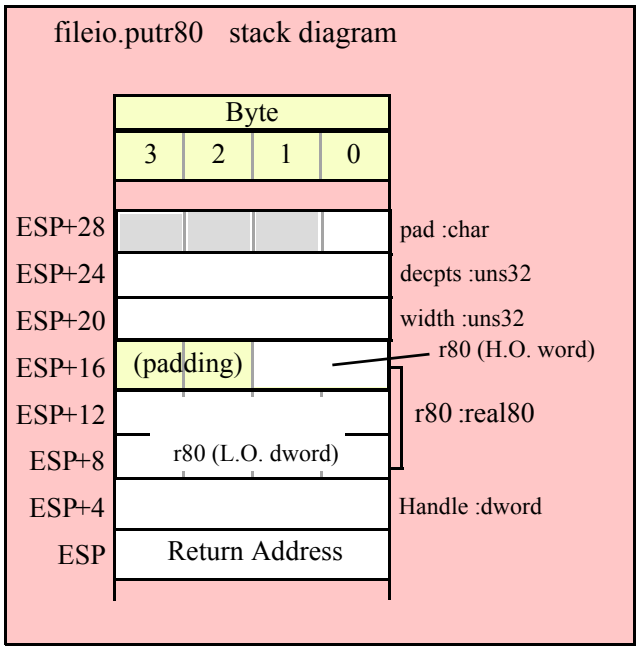
This procedure writes an 80-bit extended precision floating point value to the file as a string. The string consumes exactly width characters in the output file. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation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

HLA high-level calling sequence examples:

```
fileio.putr80( fileHandle, r80Var, width, decpts, fill );
fileio.putr80( fileHandle, r80Var, 10, 2, '*' );

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

var
  r80Temp:real80;
  .
  .
  .
fstp( r80Temp );
fileio.putr80( fileHandle, r80Temp, 12, 2, al );
```

HLA low-level calling sequence examples:

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

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

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



15.3.7 Generic File Output Routine

`fileio.put( list_of_items )`

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

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

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

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

The above is roughly equivalent to

```
fileio.puts( fileHandle, "I=" );
fileio.putu32( fileHandle, i );
fileio.puts( fileHandle, " j=" );
fileio.putu32( fileHandle, j );
fileio.newln(fileHandle);
```

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

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

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

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

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

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

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

If you specify a class variable (object) and that class defines a "toString" method, then fileio.put macro will call the associated toString method and output that string to the file. Note that the toString method must dynamically allocate storage for the string by calling stralloc. This is because fileio.put will call strfree on the string once it outputs the string.

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

15.4 File Input Routines

The HLA Standard Library provides a complementary set of file input routines. These routines behave in a fashion quite similar to the stdin.*XXXX* routines. See those routines for a additional examples of these procedures.

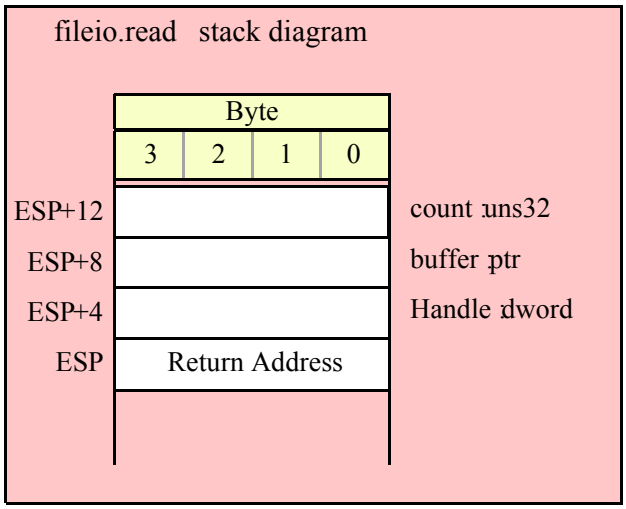
15.4.1 General File Input Routines

fileio.read(Handle:dword; var buffer:byte; count:uns32)

This routine reads a sequence of count bytes from the specified file, storing the bytes into memory at the address specified by buffer.

```
HLA high-level calling sequence examples:


fileio.read( fileHandle, buffer, count );
fileio.read( fileHandle, [eax], 1024 );



HLA low-level calling sequence examples:

// If buffer is a static variable:

push( fileHandle );
pushd( &buffer );
push( count );
call fileio.read;

// If buffer is not static, 32-bit register available:

push( fileHandle );
lea( eax, buffer );
push( eax );
push( count );
call fileio.read;

// If buffer is not static, no register available:

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

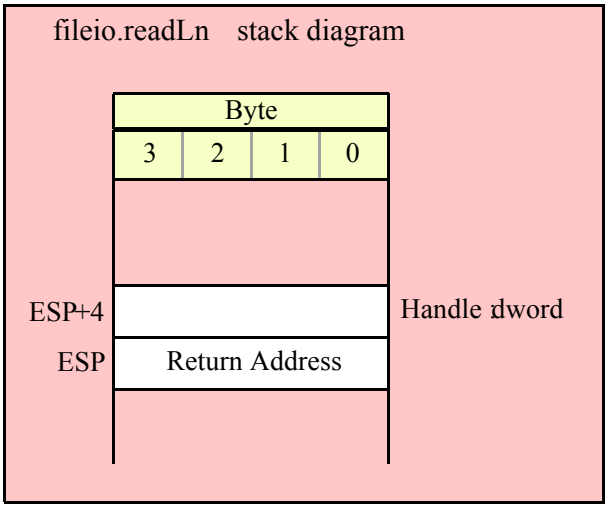
```
push( eax );
lea( eax, buffer );
mov( eax, [esp+4] );
pop( eax );
push( count );
call fileio.read;
```



`fileio.readLn( Handle:dword );`

This function reads, and discards, all characters in the file up to the next newline sequence (or end of file).

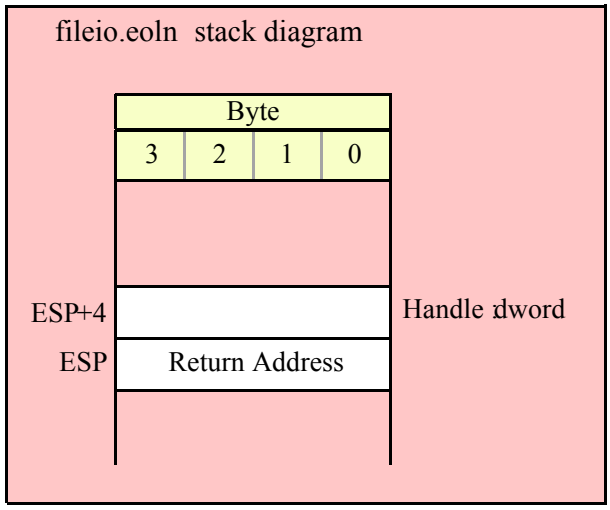
```
HLA high-level calling sequence examples:


fileio.readLn( fileHandle );


HLA low-level calling sequence examples:


push( fileHandle );
call fileio.readLn;
```



`fileio.eoln( Handle:dword ); @returns( "al" );`

This function returns true (1) in EAX if the file is at the end of a line (note that the "returns" value is "al" even though this function returns its result in all of EAX). This function eats the newline sequence from the input.

```
HLA high-level calling sequence examples:


fileio.eoln( fileHandle );
mov( al, eolnVar );

HLA low-level calling sequence examples:


push( fileHandle );
call fileio.eoln;
mov( al, eolnVar );
```

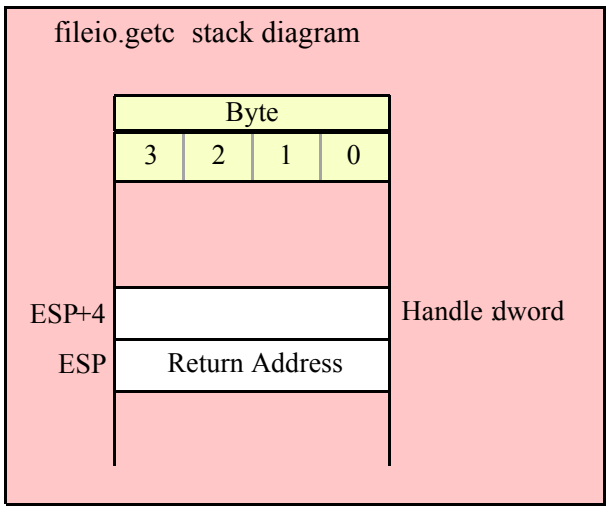


15.4.2 Character and String Input Routines

The following functions read character data from an input file specified by filevar. Note that HLA's fileio module does not provide the ability to read character set data directly from the user. However, you can always read a string and then convert that string to a character set using the appropriate function in the cset module.

`fileio.getc( Handle:dword );  @returns( "al" );`

This function reads a single character from the file and returns that chraacter in the AL register. This function assumes that the file you've opened is a text file. Note that fileio.getc does not return the end of line sequence as part of the input stream. Use the fileio.eoln function to determine when you've reached the end of a line of text. Because fileio.getc preprocesses the text file (removing end of line sequences) you should not use it to read binary data, use it only to read text files.



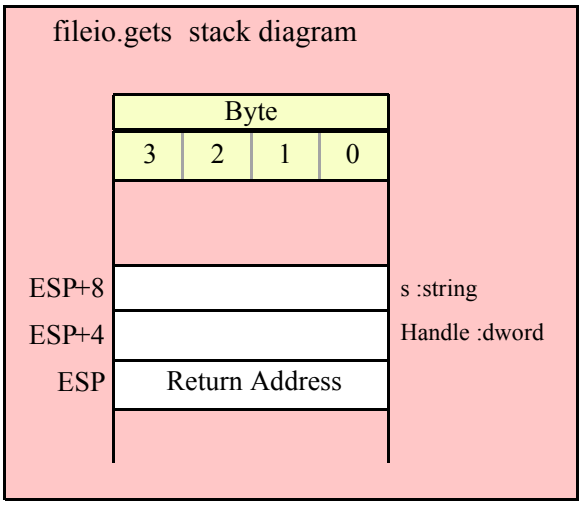
`fileio.gets( Handle:dword; s:string );`

This function reads a sequence of characters from the current file position through to the next end of line sequence and stores these characters (without the end of line sequence) into the string variable you pass as a parameter. Before calling this routine, you must allocate sufficient storage for the string. If fileio.gets attempts to read a larger string than the string's MaxStrLen value, fileio.gets raises a string overflow exception.

Note that this function does not store the end of line sequence into the string, though it does consume the end of line sequence. The next character a fileio function will read from the file will be the first character of the following line.

If the current file position is at the end of some line of text, then fileio.gets consumes the end of line and stores the empty string into the s parameter.

```
HLA high-level calling sequence examples:


fileio.gets( fileHandle, inputStr );
fileio.gets( fileHandle, eax ); // EAX contsins string value


HLA low-level calling sequence examples:


push( fileHandle );
push( inputStr );
call fileio.gets;

push( fileHandle );
push( eax );
call fileio.gets;
```

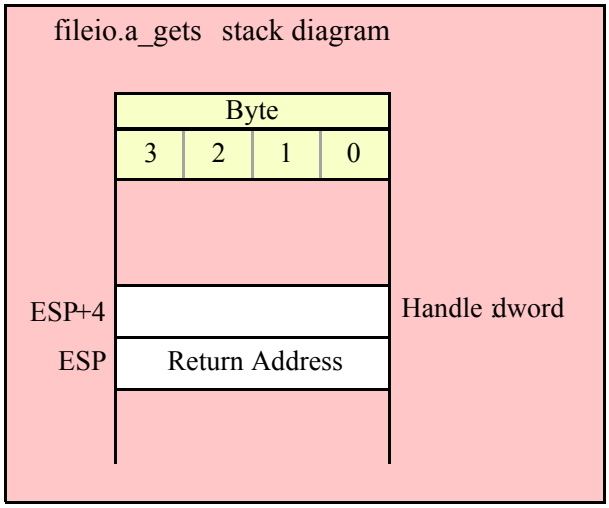


`fileio.a_gets( Handle:dword ); @returns( "eax" );`

Like fileio.gets, this function also reads a string from the file. However, rather than storing the string data into a string you supply, this function allocates storage for the string on the heap and returns a pointer to this string in the EAX register. You code should call strfree to release this storage when you're done with the string data.

The fileio.a_gets function imposes a line length limit of 1,024 characters. If this is a problem, you should modify the source code for this function to raise the limit. This functions raises an exception if you attempt to read a line longer than this internal limit.

```
HLA high-level calling sequence examples:


fileio.a_gets( fileHandle );
mov( eax, inputStr );

HLA low-level calling sequence examples:


push( fileHandle );
call fileio.a_gets;
mov( eax, inputStr );
```

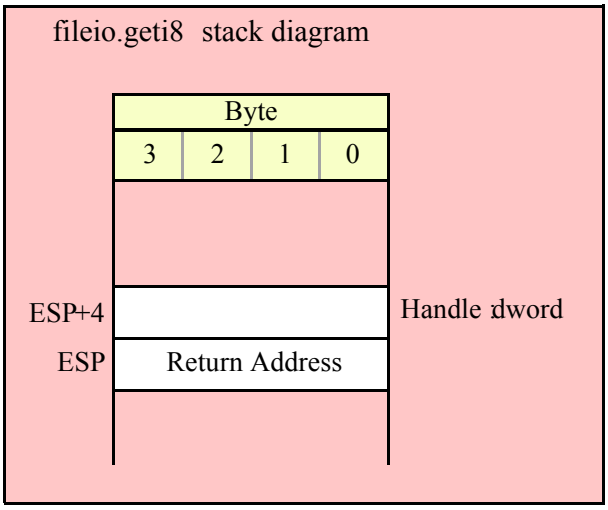


15.4.3 Signed Integer Input Routines

`fileio.geti8( Handle:dword ); @returns( "al" );`

This function reads a signed eight-bit decimal integer in the range -128..+127 from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.geti8 function raises an appropriate exception if the input violates any of these rules or the value is outside the range -128..+127. This function returns the binary form of the integer in the AL register.

```
HLA high-level calling sequence examples:


fileio.geti8( fileHandle );
mov( al, i8Var );


HLA low-level calling sequence examples:

push( fileHandle );
call fileio.geti8;
mov( al, i8Var );
```

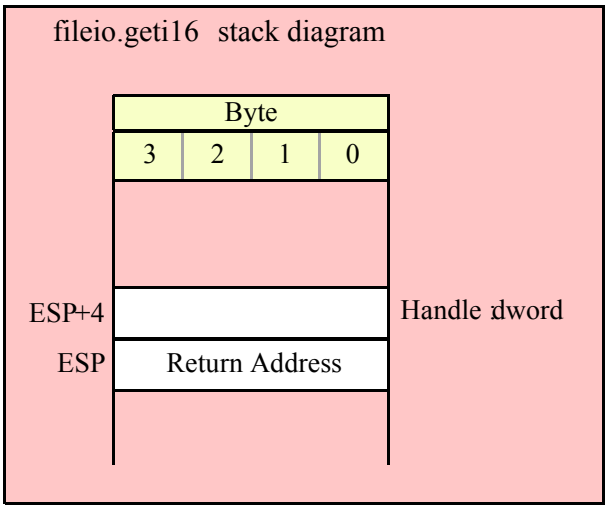


`fileio.geti16( Handle:dword ); @returns( "ax" );`

This function reads a signed 16-bit decimal integer in the range -32768..+32767 from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.geti16 function raises an appropriate exception if the input violates any of these rules or the value is outside the range -32768..+32767. This function returns the binary form of the integer in the AX register.

```
HLA high-level calling sequence examples:

fileio.geti16( fileHandle );
mov( ax, i16Var );

HLA low-level calling sequence examples:

push( fileHandle );
call fileio.geti16;
```

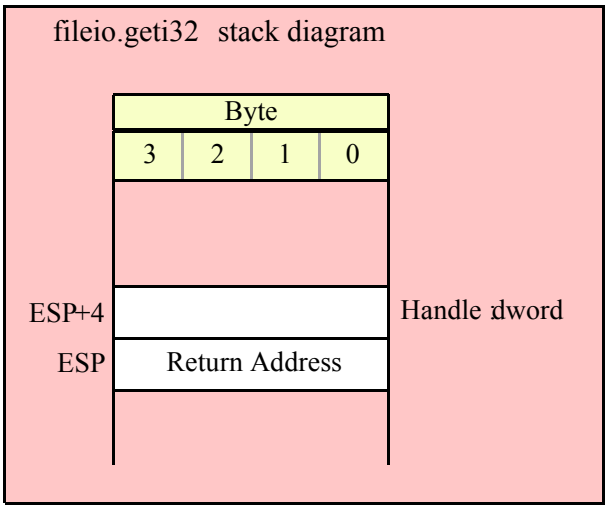
```
mov( ax, i16Var );
```



`fileio.geti32( Handle:dword ); @returns( "eax" );`

This function reads a signed 32-bit decimal integer in the (approximate) range ±2 Billion from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.geti32 function raises an appropriate exception if the input violates any of these rules or the value is outside the range plus or minus two billion. This function returns the binary form of the integer in the EAX register.

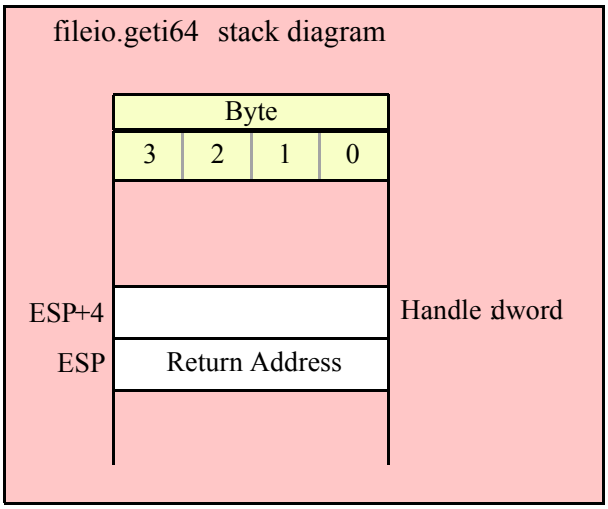
```
HLA high-level calling sequence examples:


fileio.geti32( fileHandle );
mov( eax, i32Var );

HLA low-level calling sequence examples:


push( fileHandle );
call fileio.geti32;
mov( eax, i32Var );
```



`fileio.geti64( Handle:dword );`

This function reads a signed 64-bit decimal integer from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.geti64 function raises an appropriate exception if the input violates any of these rules or the value is outside the range of a 64-bit signed integer. This function returns the 64-bit result in EDX:EAX.

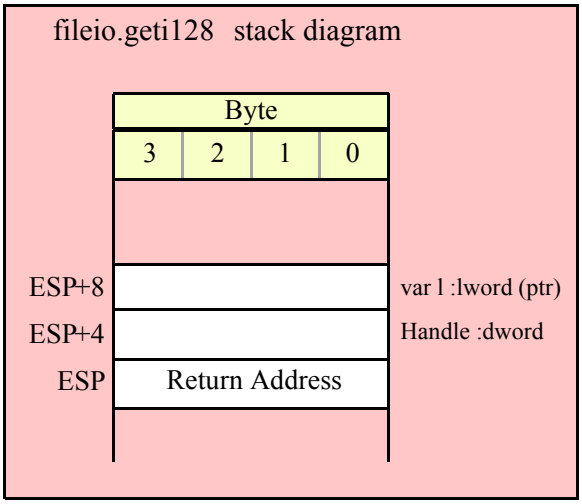
```
HLA high-level calling sequence examples:


fileio.geti64( fileHandle );
mov( edx, (type dword i64Var[4]) );
mov( eax, (type dword i64Var[0]) );


HLA low-level calling sequence examples:


push( fileHandle );
call fileio.geti64;
mov( edx, (type dword i64Var[4]) );
mov( eax, (type dword i64Var[0]) );
```



fileio.geti128(Handle:dword; var dest:lword);

This function reads a signed 128-bit decimal integer from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.geti128 function raises an appropriate exception if the input violates any of these rules or the value is outside the range of a 128-bit signed integer. This function stores the 128-bit result in the lword you pass as a reference parameter.

```
HLA high-level calling sequence examples:


fileio.geti128( fileHandle, lwordVar );


HLA low-level calling sequence examples:

// If lwordVar is a static variable:

push( fileHandle );
pushd( &lwordVar );
call fileio.geti128;

// If lwordVar is a not static variable
// and a 32-bit register is available:

push( fileHandle );
lea( eax, lwordVar ); // Assume EAX is available
push( eax);
call fileio.geti128;
```

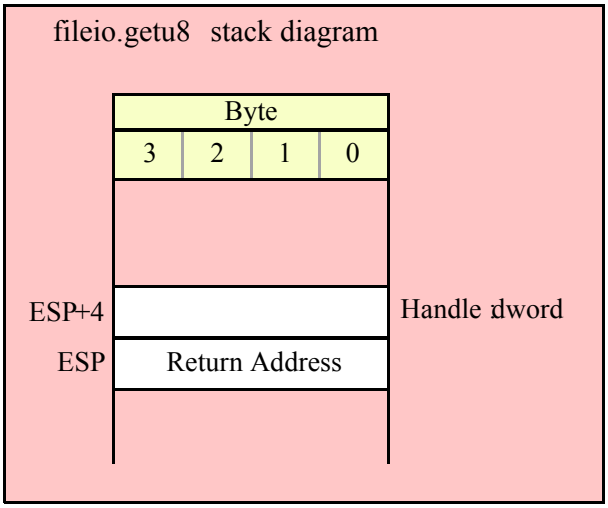


15.4.4 Unsigned Integer Input Routines

`fileio.getu8( Handle:dword ); @returns( "al" );`

This function reads an unsigned eight-bit decimal integer in the range 0..+255 from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.getu8 function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..255. This function returns the binary form of the integer in the AL register.

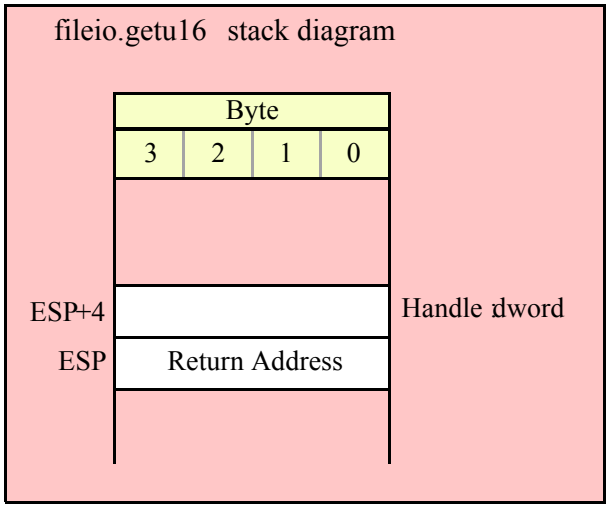
```
HLA high-level calling sequence examples:


fileio.getu8( fileHandle );
mov( al, u8Var );

HLA low-level calling sequence examples:

push( fileHandle );
call fileio.getu8;
mov( al, u8Var );
```



fileio.getu16(Handle:dword); @returns("ax");

This function reads an unsigned 16-bit decimal integer in the range 0..+65535 from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.getu16 function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..65535. This function returns the binary form of the integer in the AX register.

```
HLA high-level calling sequence examples:


fileio.getu16( fileHandle );
mov( ax, u16Var );

HLA low-level calling sequence examples:

push( fileHandle );
call fileio.getu16;
mov( ax, u16Var );
```

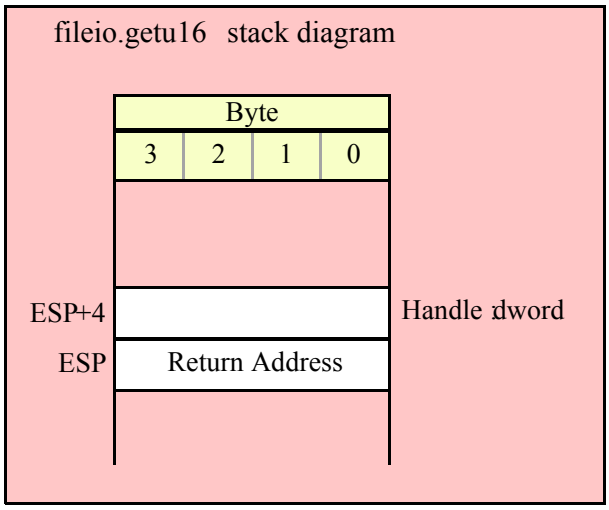


fileio.getu32(Handle:dword); @returns("eax");

This function reads an unsigned 32-bit decimal integer in the range 0..+4,294,967,295 from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.getu32 function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..4,294,967,295. This function returns the binary form of the integer in the EAX register.

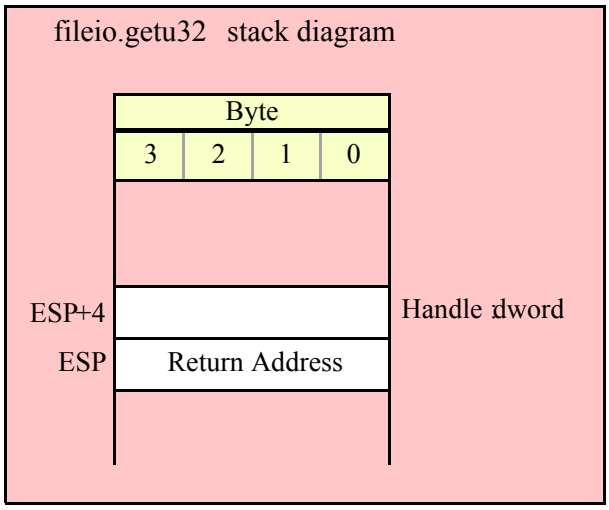
```
HLA high-level calling sequence examples:


fileio.getu32( fileHandle );
mov( eax, u32Var );

HLA low-level calling sequence examples:


push( fileHandle );
call fileio.getu32;
```

```
mov( eax, u32Var );
```



`fileio.getu64( Handle:dword );`

This function reads an unsigned 64-bit decimal integer from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.getu64 function raises an appropriate exception if the input violates any of these rules or the value is outside the range $0..2^{64}-1$. This function returns the binary form of the integer in the the EDX:EAX registers.

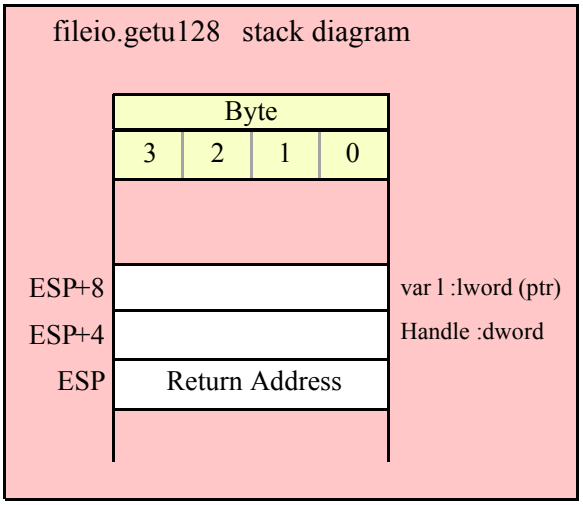


`fileio.getu128( Handle:dword; var dest:lword );`

This function reads an unsigned 128-bit decimal integer from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.getu128 function raises an appropriate exception if the input violates any of these rules or the value is outside the range $0..2^{128}-1$. This function returns the binary form of the integer in the lword parameter you pass by reference.

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

```
fileio.getu64( fileHandle );
mov( edx, (type dword u64Var[4]) );
mov( eax, (type dword u64Var[0]) );


HLA low-level calling sequence examples:


push( fileHandle );
call fileio.getu64;
mov( edx, (type dword u64Var[4]) );
mov( eax, (type dword u64Var[0]) );
```

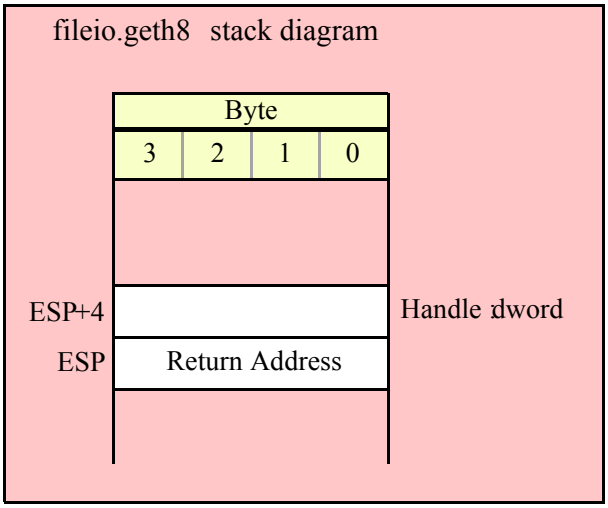


15.4.5 Hexadecimal Input Routines

`fileio.geth8( Handle:dword ); @returns( "al" );`

This function reads an eight-bit hexadecimal integer in the range 0..$FF from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.geth function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..$FF. This function returns the binary form of the value in the AL register.

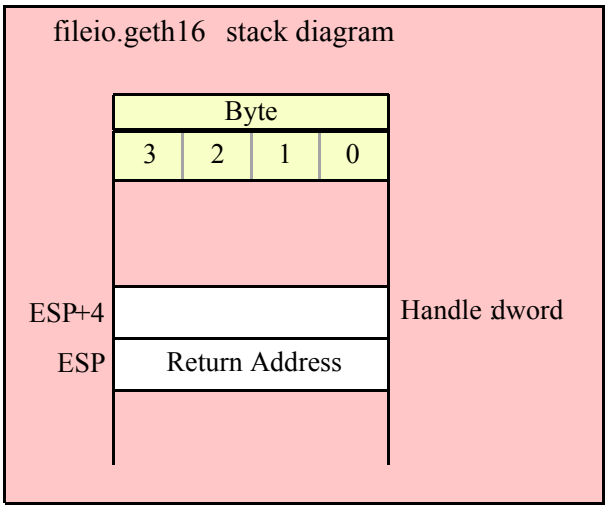
```
HLA high-level calling sequence examples:


fileio.geth8( fileHandle );
mov( al, h8Var );


HLA low-level calling sequence examples:


push( fileHandle );
call fileio.geth8;
mov( al, h8Var );
```



`fileio.geth16( Handle:dword );  @returns( "ax" );`

This function reads a 16-bit hexadecimal integer in the range 0..$FFFF from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.geth16 function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..$FFFF. This function returns the binary form of the value in the AX register (zero-extended into EAX).

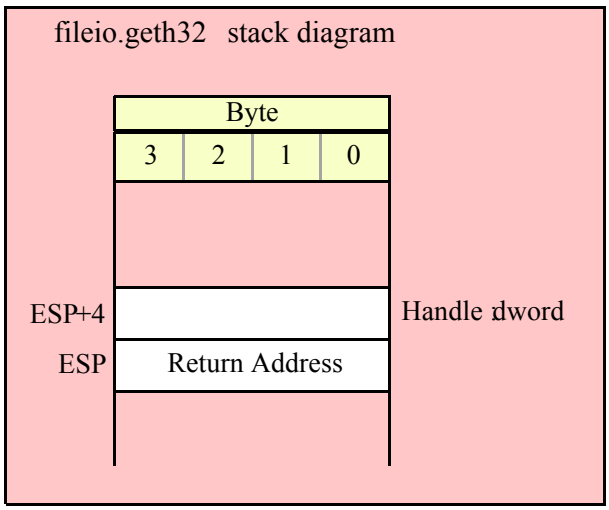
```
HLA high-level calling sequence examples:


fileio.geth16( fileHandle );
mov( ax, h16Var );

HLA low-level calling sequence examples:


push( fileHandle );
call fileio.geth16;
mov( ax, h16Var );
```



fileio.geth32(Handle:dword); @returns("eax");

This function reads a 32-bit hexadecimal integer in the range 0..$FFFF_FFFF from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.geth32 function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..$FFFF_FFFF. This function returns the binary form of the value in the EAX register.

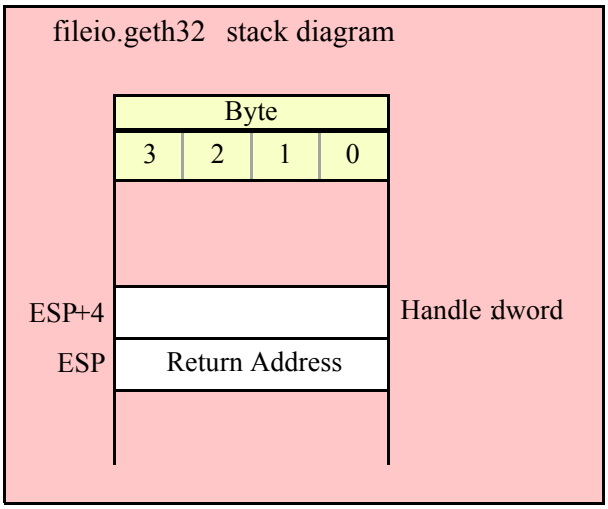
```
HLA high-level calling sequence examples:


fileio.geth32( fileHandle );
mov( eax, h32Var );

HLA low-level calling sequence examples:


push( fileHandle );
call fileio.geth32;
mov( eax, h32Var );
```



`fileio.geth64( Handle:dword );`

This function reads a 64-bit hexadecimal integer in the range 0..$FFFF_FFFF_FFFF_FFFF from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.geth64 function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..$FFFF_FFFF_FFFF_FFFF. This function returns the 64-bit result in the EDX:EAX register pair.

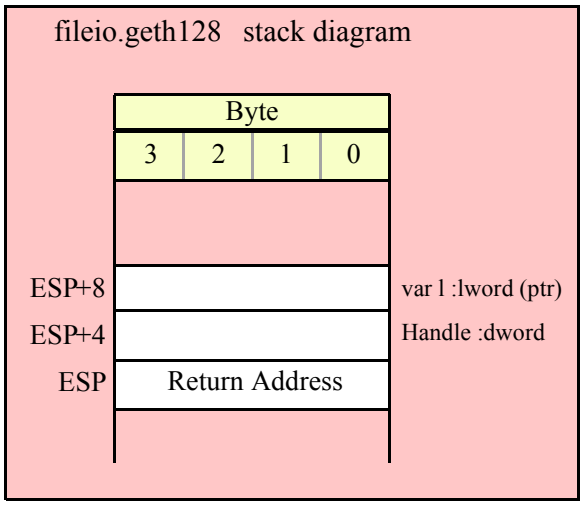
```
HLA high-level calling sequence examples:


fileio.geth64( fileHandle );
mov( edx, (type dword h64Var[4]) );
mov( eax, (type dword h64Var[0]) );

HLA low-level calling sequence examples:


push( fileHandle );
call fileio.geth64;
mov( edx, (type dword h64Var[4]) );
mov( eax, (type dword h64Var[0]) );
```



fileio.geth128(Handle:dword; var dest:lword);

This function reads a 128-bit hexadecimal integer in the range zero through $FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.geth128 function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..$FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF. This function stores the 128-bit result into the variable you pass as a reference parameter.

```
HLA high-level calling sequence examples:


fileio.geth128( fileHandle, lwordVar );


HLA low-level calling sequence examples:

// If lwordVar is a static variable:

push( fileHandle );
pushd( &lwordVar );
call fileio.geth128;

// If lwordVar is a not static variable
// and a 32-bit register is available:

push( fileHandle );
lea( eax, lwordVar ); // Assume EAX is available
push( eax);
call fileio.geth128;
```

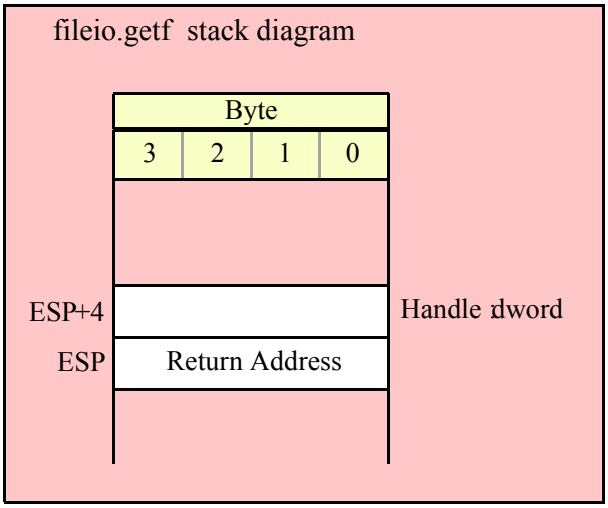


15.4.6 Floating Point Input

`fileio.getf( Handle:dword );`

This function reads an 80-bit floating point value in either decimal or scientific from from the file and leaves the result sitting on the FPU stack. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by an optional minus sign and a sequence of characters that represent a floating point value. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. This function raises an appropriate exception if an error occurs.

```
HLA high-level calling sequence examples:


fileio.getf( fileHandle );
fstp( fpVar );

HLA low-level calling sequence examples:


push( fileHandle );
call fileio.getf;
fstp( fpVar );
```



15.4.7 Generic File Input

`fileio.get( List_of_items_to_read );`

This is a macro that allows you to specify a list of variable names as parameters. The fileio.get macro reads an input value for each item in the list and stores the resulting value in each corresponding variable. This macro determines the type of each variable that you pass it and emits a call to the appropriate fileio.get*xxx* function to read the actual value. As an example, consider the following call to filevar.get:

```
fileio.get( i32, charVar, u16, strVar );
```

The macro invocation above expands into the following:

```
push( eax );
fileio.geti32( i32 );
fileio.getc();
mov( al, charVar );
fileio.geti16();
mov( ax, u16 );
fileio.gets( strVar );
pop( eax );
```

Notice that fileio.get preserves the value in the EAX and EDX registers even though various fileio.get*xxx* functions use these registers. Note that fileio.get automatically handles the case where you specify EAX as an input variable and writes the value to [esp] so that in properly modifies EAX upon completion of the macro expansion.

Note that fileio.get supports eight-bit, 16-bit, 32-bit, 64-bit, and 128-bit input values. It automatically selects the approriate input routine based on the type of the variable you specify.