14 File Class Module (fileclass.hhf)

The HLA Standard Library provides an object-oriented file access mechanism implemented via the *file_t* and *virtualfile_t* classes. Unless otherwise specified, this document will use the term "file class" to describe the generic file class rather than the specific instance of the *file_t* class (which uses static linking for all functions).

Note: HLA also provides a *fileio* library module that does file I/O using traditional procedures rather than class objects. If you're more comfortable using such a programming paradigm, or you prefer your code to be a bit more efficient, you should use the *fileio* module.

Note: Currently, the *file_t* class is implemented as a thin layer over the *fileio* module. That is, functions in the file class simply pass their parameters on to the corresponding functions in the fileio module. The ultimate intent, however, is for the *file_t* class to implement buffered I/O to improve performance. Because of the wide variety of operating systems that the HLA Standard Library supports (and will support), this may lead to some functionality limitations in future versions of the *file_t* class. In particular, you should only use *file_t* class objects to access files on block structured (disk) devices and avoid accessing character-oriented or other device types. Also, *file_t* objects will provide the best performance for sequential files. Though the intent is to fully support random-access to file data via *file_t* objects, you may get better performance by using the traditional file I/O functions in the *fileio* module.

Note: the *virtualFile_t* class is completely different from the *file_t* class. In particular, it is not a thin layer over the *fileio* module. All of the functions in the *virtualFile_t* class ultimately call the *virtualFile_t.read* and *virtualFile_t.write* functions to do file I/O. If you override these two functions (*read* and *write*), you will override the behavior of all methods in the *virtualFile_t* class. Note that this is not true for *file_t* objects.

Warning: *Don't forget that HLA objects modify the values in the ESI and EDI registers whenever you call a class procedure, method, or iterator. Do not leave any important values in either of these register when making calls to the following routines. If the use of ESI and EDI is a problem for you, you might consider using the fileio module that does not suffer from this problem.*

A Note About Thread Safety: The *file* class functions and the operating system maintain system-wide values to track things like file position within a file. Currently, these values apply to all threads in a process (and, in the case of the OS, all processes in the system). When accessing the same *file* object from different threads, you should use synchronization to serialize access to the *file* object.

Note about function overloading: the functions in the *file* classes use function overloading in order to allow you to specify the parameter lists in different ways. The macro that handles the overloading generally coerces the possible parameter types into a single object that it passes to the underlying function. The documentation for the specific functions will tell you whether a symbol is a macro or a function. For the most part, this should matter to you unless you are taking the address of a function (which you cannot do with a macro). See the HLA documentation for more details on function overloading via macros.

14.1 File Class Methods/Procedures

In most HLA classes, there are three types of functions: (static) procedures, (dynamic) methods, and dynamic iterators. The only difference between a method and a procedure is how the program actually calls the function: the program calls procedures directly, it calls methods indirectly through an entry in the virtual method table (VMT). The system always calls iterators indirectly through the VMT, so we will not consider them further. Static procedure calls are very efficient, but you lose the benefits of inheritence and functional polymorphism when you define a function as a static procedure in a class. Methods, on the other hand, fully support polymorphic calls, but introduce some efficiency issues.

First of all, unlike static procedures, your program will link in all methods defined in your program *even if you don't explicitly call those methods in your program*. Because the call is indirect, there really is no way for the assembler and linker to determine whether you've actually called the function, so it must assume that you do call it and links in the code for each method in the class. This can make your program a little larger because it may be including several date class functions that you don't actually call.

The second efficiency issue concerning method calls is that they use the EDI register to make the indirect call (static procedure calls do not disturb the value in EDI). Therefore, you must ensure that EDI is free and available before making a virtual method call, or take the effort to preserve EDI's value across such a call.

A third, though extremely minor, efficiency issue concerning methods is that the class' VMT will need an extra entry in the virtual method table. As this is only four bytes per class (not per object), this probably isn't much of a concern.

The predefined *file_t* and *virtualFile_t* objects differ in how they define the functions appearing in the class types. The *file_t* type uses static procedures for all functions, the *virtualFile_t* type uses methods for all class functions. Therefore, *file_t* object types will make direct calls to all the functions (and only link in the procedures you actually call); however, *file_t* objects do not support function polymorphism in derived classes. The *virtualFile_t* type does support polymorphism for all the class methods, but whenever you use this data type

you will link in all the methods (even if you don't call them all) and calls to these methods will require the use of the EDI register.

It is important to understand that *file_t* and *virtualFile_t* are two separate types. Neither is derived from the other. Nor are the two types compatible with one another. You should take care not to confuse objects of these two types if you're using both types in the same program.

14.2 A Quick Note

The following sections do not include sample code demonstrating the calling sequences for a couple of reasons:

For high level calls, the syntax depends on the object name and type.

Low-level calling sequences don't appear here because it doesn't really make sense to make a low-level object invocation; people wanting to make low-level calls will probably use the standard

fileio procedures rather than the object-oriented ones.

These functions are really intended for use by programmers experienced with HLA's Object-oriented assembly facilities.

For the same reasons, there are no stack diagrams for these function calls. If you want more information on making calls to HLA class methods and procedures, please consult the HLA documentation.

In the following function descriptions, the symbol <object> is used to specify a file class object or a pointer to a *file_t* class object. This wherever this document uses the name "file_t", you may substitute (as appropriate) "virtualFile_t". Note that class invocations of static procedures (e.g., "file_t.open") are illegal with the single exception of the constructor (the create procedure). If you call a file class procedure directly, the system will raise an exception (as ESI, which should be pointing at the object's data, will contain NULL).

14.3 General File Operations

The functions in this category let you initialize file objects, access fields of the file objects, and perform other housekeeping tasks.

```
<object>.create; @returns( "esi" );
file_t.create; @returns( "esi" );            [to create dynamic objects]
virtualFile_t.create; @returns( "esi" );  [to create dynamic objects]
```

The file class provides a *file_t.create* or *virtualFile_t.create* constructor which you should always call before making use of a file variable. For file variables (as opposed to file pointer variables), you should call this routine specifying the name of the file variable. For file pointer variables, you should call this routine using the class name and store the pointer returned in EAX into your file variable. For example, to initialize the two following two file objects, you would use code like the following:

```
var
  MyOutputFile: file_t;
  filePtr:     pointer to file_t;
  .
  .
  .

  MyOutputFile.create();

  file_t.create();
  mov( eax, filePtr );
```

Note that the *file_t.create* constructor simply initializes the virtual method table pointer and does other necessary internal initialization. The constructor does not open a file or perform other file-related activities.

<*object*>.handle; @returns("eax");

This function returns the file *handle* in the EAX register. The returned value is invalid if you have not opened the file. You can pass this handle value to any of the Standard Library file routines (e.g., *fileio.putc*) that expect a handle. You may also pass this value to OS API functions that expect a file handle.

```
HLA high-level calling sequence examples:

filePtr.handle();
mov( eax, fileHandle );
```

14.4 Opening and Closing Files

<*object*>.open(filename:string; access:dword)

This method opens an existing file. The *filename* parameter is a string specifying the name of the file you wish to open. The *access* parameter is one of the following:

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

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

Before accessing the data in a file, you must open the file (which initializes the file handle). The <*object*>.*open* and <*object*>.*openNew* methods are excellent tools for this purpose. You may also open the file using direct calls to the OS API, but you must initialize the <*object*>.*fileHandle* field of the class variable before making any other method calls in the file class.

```
HLA high-level calling sequence examples:

filePtr.open( "myfile.txt", fileio.r );

// 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:

MyOutputFile.open( filenameStr, accessVarByte );

filePtr.open( someStr, al );
```

<*object*>.openNew(filename:string)

This function opens a new file for writing (if the file already exists, it is first deleted and then a new file is opened for writing). The file is given the "normal" attribute.

Before accessing the data in a file, you must open the file (which initializes the file handle). The <*object*>.*open* and <*object*>.*openNew* methods are excellent tools for this purpose. You may also open the file using direct calls to the OS API, but you must initialize the <*object*>.*fileHandle* field of the class variable before making any other method calls in the file class.

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

```
filePtr.openNew( "myfile.txt" );

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

MyOutputFile.openNew( eax );
```

<*object*>.close;

This method closes a file opened via <*object*>.*open* or <*object*>.*openNew* and flushes any buffered data to the disk.

```
HLA high-level calling sequence examples:

filePtr.close();
MyOutputFile.close();
```

14.5 File Predicates

The functions in this category test conditions associated with the file.

<object>.eof(); @returns("al");

This function returns true in the AL register if the file pointer is at the end of the file. It returns false if the program can read additional data from the file.

Warning: <*object*>.*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, <*object*>.*eof*'s behavior is incorrect (it will wind up eating a character from the interactive input stream every time you call it). Unfortunately, none of the Oses that HLA supports provide 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( !filePtr.eof( someHandle )) do
.
.
.
endwhile;
```

You should write the following:

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

endtry;
```

Note: under Windows, <*object*>.*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, <*object*>.*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( !filePtr.eof( fileHandle ) ) do

  <<something while not at EOF>>

endwhile;
```

<object>.eoln(); @returns("al");

This function returns true in AL if the file pointer is currently pointing at the OS' end-of-line sequence in the file (carriage return/line feed for Windows, linefeed for other operating systems).

```
HLA high-level calling sequence examples:


filePtr.eoln();
```

14.6 Miscellaneous Output

The following file output routines all assume that you've opened the <object> file variable via a call to *<object>.open* and you've successfully opened the file for output.

***<object>*.write(var buffer:var; count:dword)**

This method writes the number of bytes specified by the *count* parameter 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: Notice that the *buffer* parameter is an untyped reference parameter. Untyped reference parameters have special properties, so be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) if you are not absolutely sure you understand how untyped reference parameters operate.

```
HLA high-level calling sequence examples:

filePtr.write( buffer, count );

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

filePtr.write( val bufPtr, count );

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

filePtr.write( bufPtr, 4 );
```

***<object>*.putbool(b:boolean);**

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

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

```
filePtr.putbool( boolVar );

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

MyOutputFile.putbool( al );
```

***<object>*.newln( );**

This function writes a newline sequence (carriage return/line feed under Windows, linefeed under other operating systems) to the specified output file (<*object*>).

```
HLA high-level calling sequence examples:

filePtr.newln();
MyOutputFile.newln();
```

14.7 Character, Character Set, and String Output

The following file output routines all assume that you've opened the <object> file variable via a call to <*object*>.*open* and you've successfully opened the file for output.

<*object*>.**putc(c:char)**

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

```
HLA high-level calling sequence examples:

filePtr.putc( charVar );

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

MyOutputFile.putc( al );
```

<*object*>.**putcSize(c:char; width:int32; fill:char)**

Outputs the character *c* to the file filevar 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 <*object*>.*putcSize* writes *c* left justfied in a field of *width* characters; if *width* is a negative value less than one, then <*object*>.*putcSize* writes *c* right justified in a field of *width* characters.

```
HLA high-level calling sequence examples:

filePtr.putcSize( charVar, width, padChar );
```

<*object*>.**putcset(cst:cset);**

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

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

```
filePtr.putcset( csVar );
MyOutputFile.putcset( [ebx] ); // EBX points at the cset.
```

<*object*>.**puts(s:string);**

This procedure writes the value of the string parameter to the file.

```
HLA high-level calling sequence examples:

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

<*object*>.**putsSize(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 <*object*>.*puts*. On the other hand, if the absolute value of *width* is greater than the length of *s*, then <*object*>.*putsSize* writes *width* characters to the output file. This procedure emits the *fill* character in the extra print positions. If *width* is positive, then <*object*>.*putsSize* right justifies the string in the print field. If *width* is negative, then <*object*>.*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:

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

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

MyOutputFile.putsSize( ebx, ecx, al );

filePtr.putsSize( "Hello World", 25, padChar );
```

14.8 Hexadecimal Numeric Output

The following file output routines all assume that you've opened the <object> file variable via a call to <*object*>.*open* and you've successfully opened the file for output.

<*object*>.**putb(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:


filePtr.putb( byteVar );

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

```
MyOutputFile.putb( al );
```

<*object*>.**puth8(b:byte);**

This procedure writes the value of *b* to the file using one or two hexadecimal digits (the minimum necessary).

```
HLA high-level calling sequence examples:

filePtr.puth8( byteVar );

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

MyOutputFile.puth8( al );
```

<*object*>.**puth8Size(b:byte; width:dword; fill:char)**

This procedure writes the value of *b* to the file using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <*object*>.*putcSize* function for more details on their behavior.

```
HLA high-level calling sequence examples:

filePtr.puth8Size( byteVar, width, padChar );
```

<*object*>.**putw(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:

filePtr.putw( wordVar );

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

MyOutputFile.putw( ax );
```

<*object*>.**puth16(w:word);**

This procedure writes the value of *w* to the file using 1-4 hexadecimal digits (the minimum necessary).

```
HLA high-level calling sequence examples:

filePtr.puth16( wordVar );

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

```
MyOutputFile.puth16( ax );
```

<*object*>.**puth16Size(w:word; width:dword; fill:char)**

This procedure writes the value of *w* to the file using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <*object*>.*putcSize* function for more details on their behavior.

```
HLA high-level calling sequence examples:

filePtr.puth16Size( wordVar, width, padChar );
```

<*object*>.**putd(dw:dword);**

This procedure writes the value of *d* to the file using exactly eight hexadecimal digits (including leading zeros if necessary). If the stdlib global underscores value (see the conversions module for details) contains true, then this function will also print an underscore between the fourth and fifth digits.

```
HLA high-level calling sequence examples:


filePtr.putd(dwordVar );

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

MyOutputFile.putd( eax );
```

<*object*>.**puth32(dw:dword);**

This procedure writes the value of *d* to the file using the minimum number of hexadecimal required. If the stdlib global underscores value (see the conversions module for details) contains true, then this function will also print an underscore between the fourth and fifth digits (if there are at least five digits in the number).

```
HLA high-level calling sequence examples:


filePtr.puth32( dwordVar );

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

MyOutputFile.puth32( eax );
```

<*object*>.**puth32Size(d:dword; width:dword; fill:char)**

This procedure writes the value of d to the file using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <*object*>.*putcSize* function for more details on their behavior.

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

```
filePtr.puth32Size( dwordVar, width, ' ' );

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

MyOutputFile.puth32Size( eax, width, cl );
```

<*object*>.**putq(q:qword);**

This procedure writes the value of *q* to the file using exactly 16 hexadecimal digits (including leading zeros if necessary). If the stdlib global underscores value (see the conversions module for details) contains true, then this function will also print an underscore between each group of four digits.

```
HLA high-level calling sequence example:


filePtr.putq( qwordVar );
```

<*object*>.**puth64(q:qword);**

This procedure writes the value of *q* to the file using 1-16 hexadecimal digits (the minimum necessary). If the stdlib global underscores value (see the conversions module for details) contains true, then this function will also print an underscore between each group of four digits.

```
HLA high-level calling sequence example:


MyOutputFile.puth64( qwordVar );
```

<*object*>.**puth64Size(q:qword; width:dword; fill:char)**

This procedure writes the value of *q* to the file using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <*object*>.*putcSize* function for more details on their behavior.

```
HLA high-level calling sequence example:


MyOutputFile.puth64Size( qwordVar, width, ' ' );
```

<*object*>.**puttb(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:


filePtr.puttb( tbyteVar );
```

<*object*>.**puth80(tb:tbyte)**

This procedure writes the value of *tb* to the file using 1-20 hexadecimal digits (the minimum necessary) and an intervening underscores if underscore output is enabled).

```
HLA high-level calling sequence examples:


filePtr.puth80( tbyteVar );
```

<*object*>.**puth80Size(tb:tbyte; width:dword; fill:char)**

This procedure writes the value of *tb* to the file using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <*object*>.*putcSize* function for more details on their behavior.

```
HLA high-level calling sequence examples:


filePtr.puth80Size( tbyteVar, width, ' ' );
```

<*object*>.**putl(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:


MyOutputFile.putl( lwordVar );
```

<*object*>.**puth128(l:lword)**

This procedure writes the value of *l* to the file using 1-32 hexadecimal digits (the minimum necessary) and an intervening underscores if underscore output is enabled).

```
HLA high-level calling sequence examples:


filePtr.puth128( lwordVar );
```

<*object*>.**puth128Size(l:lword; width:dword; fill:char)**

This procedure writes the value of *l* to the file using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <*object*>.*putcSize* function for more details on their behavior.

```
HLA high-level calling sequence examples:


MyOutputFile.puth128Size( tbyteVar, width, ' ' );
```

14.9 Signed Integer Numeric Output

The following file output routines all assume that you've opened the <object> file variable via a call to <*object*>.*open* and you've successfully opened the file for output.

These routines convert signed integer values to string format and write that string to the *filevar* file. The *<object>.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.

xxxSize(value, width, fill);

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

f	f	f	V	A	L	U	E

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

V	A	L	U	E	f	f	f

<*object*>.puti8 (b:byte ;

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

```
HLA high-level calling sequence examples:


filePtr.puti8( byteVar );

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

MyOutputFile.puti8( al );
```

<*object*>.puti8Size (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:

filePtr.puti8Size( byteVar, width, padChar );
```

<*object*>.puti16(w:word);

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

```
HLA high-level calling sequence examples:


MyOutputFile.puti16( wordVar );

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

filePtr.puti16( ax );
```

<*object*>.puti16Size(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:

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

<*object*>.puti32(d:dword);

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

```
HLA high-level calling sequence examples:


filePtr.puti32( dwordVar );

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

MyOutputFile.puti32( eax );
```

<*object*>.puti32Size(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:


MyOutputFile.puti32Size( dwordVar, width, ' ' );

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

filePtr.puti32Size( eax, width, cl );
```

<*object*>.puti64(q:qword);

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

```
HLA high-level calling sequence examples:


MyOutputFile.puti64( qwordVar );
```

<*object*>.puti64Size(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:

filePtr.puti64Size( qwordVar, width, ' ' );
```

<*object*>.puti128(l:lword);

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

```
HLA high-level calling sequence examples:

MyOutputFile.puti128( lwordVar );
```

<*object*>.puti128Size(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:

filePtr.puti128Size( lwordVar, width, ' ' );
```

14.10 Unsigned Integer Numeric Output

These routines convert unsigned integer values to string format and write that string to the file. The *<object>.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.

<*object*>.putu8 (b:byte)

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

```
HLA high-level calling sequence examples:

filePtr.putu8( byteVar );

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

MyOutputFile.putu8( al );
```

<*object*>.putu8size(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:

filePtr.putu8Size( byteVar, width, padChar );
```

<*object*>.putu16(w:word)

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

```
HLA high-level calling sequence examples:


filePtr.putu16( wordVar );

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

MyOutputFile.putu16( ax );
```

<*object*>.putu16size(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:

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

<*object*>.putu32(d:dword)

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

```
HLA high-level calling sequence examples:


filePtr.putu32( dwordVar );

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

MyOutputFile.putu32( eax );
```

<*object*>.putu32Size(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:


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

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

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

<*object*>.putu64(q:qword)

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

```
HLA high-level calling sequence examples:


filePtr.putu64( qwordVar );
```

<*object*>.putu64Size(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:


MyOutputFile.putu64Size( qwordVar, width, ' ' );
```

<*object*>.putu128(l:lword)

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

```
HLA high-level calling sequence examples:


filePtr.putu128( lwordVar );
```

<*object*>.putu128Size(l:lword; width:int32; fill:char);

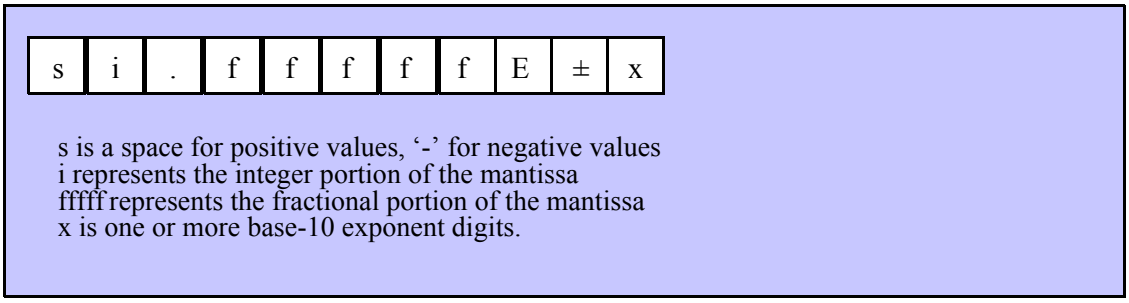
This function writes the unsigned 128-bit value you pass to the specified output file using the *width* and *fill* values as specified above.

```
HLA high-level calling sequence examples:


MyOutputFile.putu128Size( lwordVar, width, ' ' );
```

14.11 Floating-Point Numeric 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 filevar 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 *<object>.pute80*, *<object>.pute64*, and *<object>.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:



<*object*>.pute32(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:


MyOutputFile.pute32( r32Var, width );

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

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

<*object*>.pute64(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:


filePtr.pute64( r64Var, width );

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

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

<*object*>.pute80(r:real80; width:uns32)

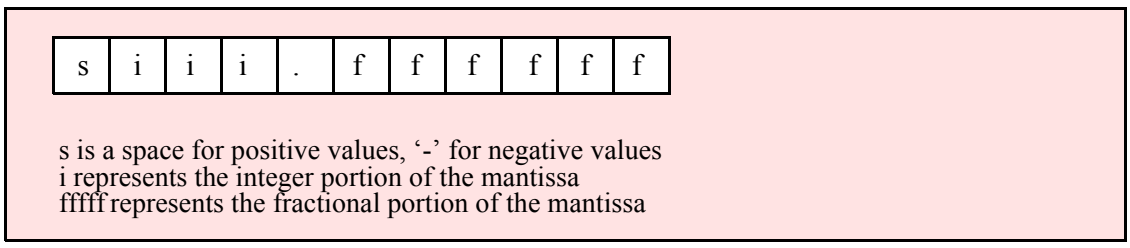
This function writes the 80-bit extended precision floating point value passed in *r* to the file using scientific/exponential notation. This procedure prints the value using *width* print positions in the file. *width* should have a minimum value of five for real numbers in the range 1e-9..1e+9 and 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:


filePtr.pute80( r80Var, width );

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

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

14.12 Floating-Point Numeric 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 file class 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 four parameters: the real value to convert, the width of the converted value, the number of digit positions to the right of the decimal point, and a padding character. The second variety only requires the first three parameters and assumes the padding character is a space. These functions write their values using the following string format:



<*object*>.putr32(r:real32; width:uns32; decpts:uns32; fill:char)

This procedure writes a 32-bit single 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 *fill* value as the padding character to fill the output with *width* characters.

HLA high-level calling sequence examples:

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

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

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

<*object*>.putr64(r:real64; width:uns32; decpts:uns32; fill:char)

This procedure writes a 64-bit double 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 *fill* as the padding character to fill the output with *width* characters.

HLA high-level calling sequence examples:

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

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

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

<*object*>.putr80(r:real80; width:uns32; decpts:uns32; fill: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 *fill* as the padding character to fill the output with *width* characters.

HLA high-level calling sequence examples:

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

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

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

14.13 Generic File Output

<object>.put(parameter_list)

<*object*>*.put* is a macro that automatically invokes an appropriate <*object*> output routine based on the type of the parameter(s) you pass it. This is a very convenient output routine and is probably the file class 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 <*object*>*.puti32,* <*object*>*.puts,* etc.

<*object*>*.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, <*object*>*.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 an <*object*>*.put* :

```
<object>.put( "I=", i, " j=", j, nl );
```

The above is roughly equivalent to

```
<object>.puts( "I=" );
<object>.puti32( i );
<object>.puts( " j=" );
<object>.puti32( j );
<object>.newln();
```

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

The <*object*>*.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:

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

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

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

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

The <*object*>*.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, tbyte, lword).

If you specify a class variable (object) and that class defines a *toString* method, the <*object*>*.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 *str.alloc*. This is because <*object*>*.put* will call *str.free* on the string once it outputs the string.

There is a known "design flaw" in the *<object>.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 *<object>.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 *<object>.put* to print it. Of course, there is no problem using the other *<object>.putXXXX* functions to display non-local VAR objects, so you can use those as well.

Important(!), don't forget that method calls (e.g., the routines that *<object>.put* translates into) modify the values in the ESI and EDI registers. Therefore, it never makes any sense to attempt to print the values of ESI and EDI within the parameter list. All you will wind up doing is printing the address of the file variable (ESI) or the address of its virtual method table (EDI). If you need to write these two values to a file, move them to another register or a memory location first.

14.14 Generic File Input

The following file input routines behave just like their standard input and file input counterparts (unless otherwise noted):

***<object>*.read(var buffer:var; count:dword)**

This function reads *count* bytes from the file and stores them into memory starting with the first byte of the *buffer* variable. This routine does not do any range checking. It is your responsibility to ensure that *buffer* is large enough to hold the data read.

Note: Notice that the *buffer* parameter is an untyped reference parameter. Untyped reference parameters have special properties, so be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) if you are not absolutely sure you understand how untyped reference parameters operate.

```
HLA high-level calling sequence examples:


MyInputFile.read( buffer, count );
MyInputFile.read( [eax], 1024 );
```

<object>.readln;
This function reads and discards all characters up to and including the newline sequence in the file.

```
HLA high-level calling sequence examples:

filePtr.readLn();
```

14.15 Character and String Input

The following functions read character data from an input file. Note that HLA's file class 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.

***<object>*.getc; @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 *<object>.getc* does not return the end of line sequence as part of the input stream. Use the *<object>..eoln* function to determine when you've reached the end of a line of text. Because *<object>..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.

```
HLA high-level calling sequence examples:

filePtr.getc();
```

```
mov( al, charVar );
```

<*object*>.gets(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 <*object*>.*gets* attempts to read a larger string than the string's *MaxLen* value, <*object*>.*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 file class 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 <*object*>.*gets* consumes the end of line and stores the empty string into the *s* parameter.

```
HLA high-level calling sequence examples:


filePtr.gets( inputStr );
filePtr.gets( eax ); // EAX contains string value
```

<*object*>.a_gets; @returns("eax");

Like <*object*>.*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 *str.free* to release this storage when you're done with the string data.

The <*object*>.*a_gets* function imposes a line length limit of 4,096 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:


MyInputFile.a_gets();
mov( eax, inputStr );
```

14.16 Signed Integer Input

The functions in this group read numeric values from the file using a signed decimal integer format. These functions read the string data, translate it to numeric form, and return that numeric data in an appropriate location.

<*object*>.geti8; @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 <*object*>.*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:


filePtr.geti8();
```

```
mov( al, i8Var );
```

<*object*>.geti16; @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 *<object>.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:

```
filePtr.geti16();
mov( ax, i16Var );
```

<*object*>.geti32; @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 *<object>.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:

```
filePtr.geti32();
mov( eax, i32Var );
```

<*object*>.geti64; @returns("edx:eax");

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 <object>.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 the EDX:EAX register pair (it returns the H.O. dword in EDX and the L.O. dword in EAX).

HLA high-level calling sequence examples:

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

***<object>*.geti128(var l: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 *<object>.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 into the lword you pass as a reference parameter.

```
HLA high-level calling sequence examples:


filePtr.geti128( lwordVar );
```

14.17 Unsigned Integer Input

The functions in this group read numeric values from the file using a signed decimal integer format. These functions read the string data, translate it to numeric form, and return that numeric data in an appropriate location.

***<object>*.getu8; @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 *<object>.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:


MyInputFile.getu8();
mov( al, u8Var );
```

***<object>*.getu16; @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 *<object>.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:


filePtr.getu16();
mov( ax, u16Var );
```

***<object>*.getu32; @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 *<object>.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:


filePtr.getu32();
mov( eax, u32Var );
```

<*object*>.getu64; @returns("edx:eax");

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 *<object>.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 EDX:EAX register pair (EDX contains the H.O. dword, EAX holds the L.O. dword).

```
HLA high-level calling sequence examples:


filePtr.getu32();
mov( eax, (type dword u64Var) );
mov( edx, (type dword u64Var[4]) );
```

<*object*>.getu128(var l: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 *<object>.getu64* 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.getu128( u128Var );
```

14.18 Hexadecimal Input

<*object*>.geth8; @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 *<object>.geth8* 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:


filePtr.geth8();
```

```
mov( al, h8Var );
```

<*object*>.geth16; @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 *<object>.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.

HLA high-level calling sequence examples:

```
MyInputFile.geth16();
mov( ax, h16Var );
```

<*object*>.geth32; @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 *<object>.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:

```
filePtr.geth32();
mov( eax, h32Var );
```

<*object*>.geth64; @returns("edx:eax");

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 *<object>.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 (EDX contains the H.O. dword, EAX contains the L.O. dword).

HLA high-level calling sequence examples:

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

<*object*>.geth128(var l:lword);

This function reads a 128-bit hexadecimal integer in the range 0..$FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" 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 *<object>.getq* 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:


filePtr.geth128( lwordVar );
```

14.19 Floating-Point Input

<*object*>.getf; @returns("st0");

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:


filePtr.getf();
fstp( fpVar );
```

14.20 Generic File Input

<object>.get(List_of_items_to_read);

This is a macro that allows you to specify a list of variable names as parameters. The *<object>.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 *<object>.getxxx* function to read the actual value. As an example, consider the following call to *<object>.get*:

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

The macro invocation above expands into the following:

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

Notice that <*object*>.*get* preserves the value in the EAX register even though various <*object*>.*getxxx* functions use this register. Note that <*object*>.*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 <*object*>.*get* only supports eight-, sixteen-, and thirty-two bit integer input. If you need to read 64-bit or 128-bit values, you must use the appropriate <*object*>.*getx64* or <*object*>.*getx128* function to achieve this.