28 The Standard Input Module (stdin.hhf)

This unit contains routines that read data from the standard input device.

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter. Whenever you request input, by calling one of the following input routines, the Standard Library routines first check to see if there is any data available in an internal buffer. If so, the routines read the data from the buffer; if not, the routines fill the buffer by reading a line of text from the Standard Input Device. Once a line is read, the routine will read its data from the newly acquired buffer. Additional calls to the standard input routines continue to read their data from this same buffer until the input line is exhausted, at which point the library routines will read more data from the Standard Input Device.

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

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).

28.1 Conversion Format Control

When reading numeric data from the standard input, the stdin 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.

28.2 File I/O Routines and the Standard Output Handle

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

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

This routine returns the handle of the Standard Input Device in the EAX register.

28.3 Standard Input Routines

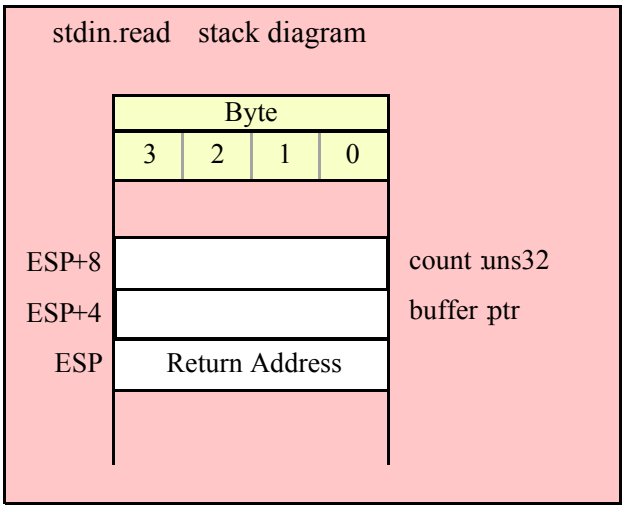
The HLA Standard Library provides a complementary set of standard input routines. These routines behave in a fashion quite similar to the stdin.*XXXX* routines. See those routines for additional examples of these procedures.

28.4 General Standard Input Routines

`stdin.read( var buffer:byte; count:uns32 )`

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

```
HLA high-level calling sequence examples:


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



HLA low-level calling sequence examples:

// If buffer is a static variable:

pushd( &buffer );
push( count );
call stdin.read;

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

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

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

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



stdin.readLn;

This routine flushes the current input buffer and immediately reads a new line of text from the user.

```
HLA high-level calling sequence examples:


stdin.readLn();


HLA low-level calling sequence examples:


call stdin.readLn;
```

stdin.eoln; @returns("al");
stdin.eoln2; @returns("al");

These functions return true if the input buffer is at the end of the current line. The stdin.eoln2 function will first remove any delimiter characters from the input buffer before testing for the end of the current line. These functions return true (1) or false (0) in the AL/EAX register.

These functions do not force a new line of input on the next stdin.get*XX* operation. I.e., if you read a string after stdin.eoln returns true, you will get the empty string as the result. Call stdin.readln to force the input of a new line.

```
HLA high-level calling sequence examples:


stdin.eoln();
mov( al, eolnVar );

HLA low-level calling sequence examples:


call stdin.eoln;
mov( al, eolnVar );
```

stdin.flushInput;

This routine flushes the internal buffer. The next call to a Standard Library input routine will force the system to read a new line of text from the user. All current data in the internal input buffer is lost.

Please note that this routine does not immediately force the input of a new line of text from the user unless the internal buffer is already empty. If the internal buffer is empty and you call this routine, it will read a new line of text from the user and then flush this text from the internal buffer.

```
HLA high-level calling sequence examples:

stdin.flushInput();

HLA low-level calling sequence examples:

call stdin.flushInput;
```

28.5 Character and String Input Routines

The following functions read character data from an input file specified by filevar. Note that HLA's stdin 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.

stdin.peekc; @returns("al");

This routine returns the character character from the standard input device without actually "reading" that character. That is, after a call to stdin.peekc, the next call to stdin.getc will return the same character as the one stdin.peekc returns. A call to stdin.peekc does not force the input of a new line of text. If the current input buffer is empty, calls to stdin.peekc return zero in the AL register. This routine returns the character in the AL register and it returns zeros in the upper three bytes of EAX.

stdin.getc(); @returns("al");

This function reads a single character from the standard inpu device and returns that character in the AL register.

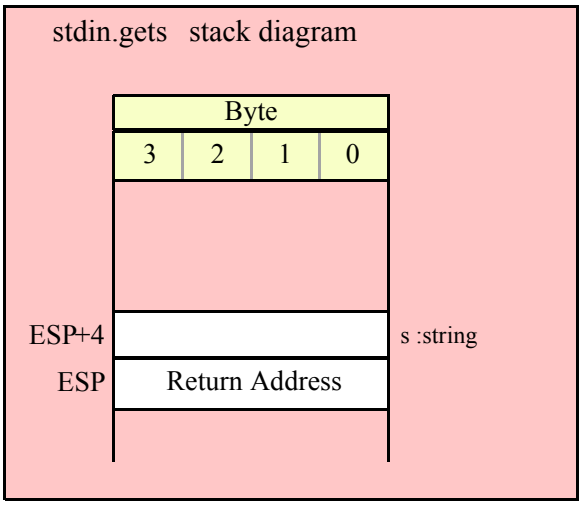
stdin.gets(s:string);

This function reads a sequence of characters from the standard input 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 stdin.gets attempts to read a larger string than the string's MaxStrLen value, stdin.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 stdin function will read from the standard input will be the first character of the following line.

If the stasndard input is at the end of some line of text, then stdin.gets consumes the end of line and stores the empty string into the s parameter.

```
HLA high-level calling sequence examples:


stdin.gets( inputStr );
stdin.gets( eax ); // EAX contsins string value


HLA low-level calling sequence examples:


push( inputStr );
call stdin.gets;

push( eax );
call stdin.gets;
```



stdin.a_gets(); @returns("eax");

Like stdin.gets, this function also reads a string from the standard input. 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 stdin.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 function raises an exception if you attempt to read a line longer than this internal limit.

```
HLA high-level calling sequence examples:


stdin.a_gets();
mov( eax, inputStr );

HLA low-level calling sequence examples:


call stdin.a_gets;
mov( eax, inputStr );
```

28.6 Hexadecimal Input Routines

The hexadecimal input routines read a numeric value from the standard input in hexadecimal format. Except for stdin.geth128, they return their results in one (or two) registers (stdin.geth128 returns its value in a pass-by-reference parameter).

stdin.geth8(); @returns("al");

This function reads an eight-bit hexadecimal integer in the range 0..$FF from the standard input device. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The stdin.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 (zero extended into EAX, so you may use EAX if it is more convenient to so so).

```
HLA high-level calling sequence examples:


stdin.geth8();
mov( al, h8Var );

HLA low-level calling sequence examples:


call stdin.geth8;
mov( al, h8Var );
```

stdin.geth16(); @returns("ax");

This function reads a 16-bit hexadecimal integer in the range 0..$FFFF from the standard input. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The stdin.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:


stdin.geth16();
mov( ax, h16Var );

HLA low-level calling sequence examples:


call stdin.geth16;
mov( ax, h16Var );
```

stdin.geth32(); @returns("eax");

This function reads a 32-bit hexadecimal integer in the range 0..$FFFF_FFFF from the standard input device. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The stdin.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:


stdin.geth32();
mov( eax, h32Var );

HLA low-level calling sequence examples:


call stdin.geth32;
mov( eax, h32Var );
```

stdin.geth64();

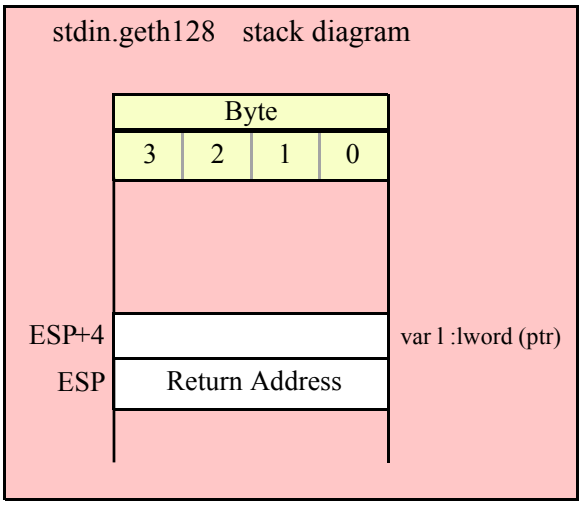
This function reads a 64-bit hexadecimal integer in the range 0..$FFFF_FFFF_FFFF_FFFF from the standard input device. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The stdin.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:


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

HLA low-level calling sequence examples:


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

stdin.geth128(var dest:lword);

This function reads a 128-bit hexadecimal integer in the range zero through $FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF from the standard input device. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The stdin.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:


stdin.geth128( lwordVar );

HLA low-level calling sequence examples:

// If lwordVar is a static variable:

pushd( &lwordVar );
call stdin.geth128;

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

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



28.7 Signed Integer Input Routines

stdin.geti8(); @returns("al");

This function reads a signed eight-bit decimal integer in the range -128..+127 from the standard input. 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. This function allows underscores in the interior of the number. The stdin.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 (signed extended into EAX).

```
HLA high-level calling sequence examples:


stdin.geti8();
mov( al, i8Var );

HLA low-level calling sequence examples:


call stdin.geti8;
mov( al, i8Var );
```

stdin.geti16(); @returns("ax");

This function reads a signed 16-bit decimal integer in the range -32768..+32767 from the standard input device. 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. This function allows underscores in the interior of the number. The stdin.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 (signed extended into EAX).

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

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

HLA low-level calling sequence examples:

```
call stdin.geti16;
mov( ax, i16Var );
```

stdin.geti32(); @returns("eax");

This function reads a signed 32-bit decimal integer in the (approximate) range ±2 Billion from the standard input device. 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. This function allows underscores in the interior of the number. The stdin.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:

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

HLA low-level calling sequence examples:

```
call stdin.geti32;
mov( eax, i32Var );
```

stdin.geti64(); @returns("edx:eax");

This function reads a signed 64-bit decimal integer from the standard input device. 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. This function allows underscores in the interior of the number. The stdin.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:

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

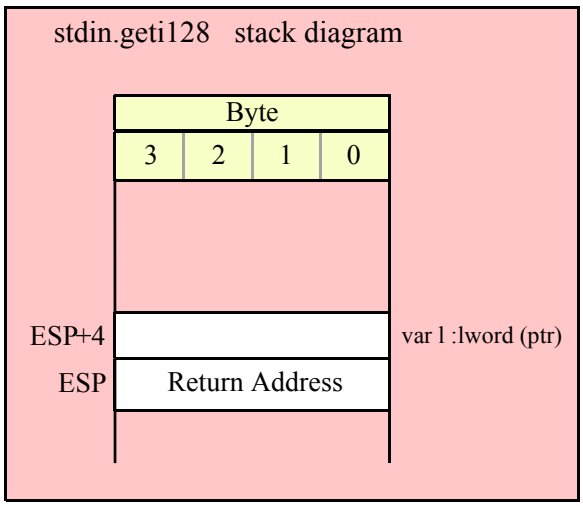
HLA low-level calling sequence examples:

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

stdin.geti128(var dest:lword);

This function reads a signed 128-bit decimal integer from the standard input. 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. This function allows underscores in the interior of the number. The stdin.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:


stdin.geti128( lwordVar );

HLA low-level calling sequence examples:

// If lwordVar is a static variable:

pushd( &lwordVar );
call stdin.geti128;

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

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



28.8 Unsigned Integer Input Routines

stdin.getu8(); @returns("al");

This function reads an unsigned eight-bit decimal integer in the range 0..+255 from the standard input device. 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. This function allows underscores in the interior of the number. The stdin.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 (zero extended into EAX).

```
HLA high-level calling sequence examples:


stdin.getu8();
mov( al, u8Var );

HLA low-level calling sequence examples:


call stdin.getu8;
mov( al, u8Var );
```

stdin.getu16(); @returns("ax");

This function reads an unsigned 16-bit decimal integer in the range 0..+65535 from the standard input device. 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. This function allows underscores in the interior of the number. The stdin.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 (zero extended into EAX).

```
HLA high-level calling sequence examples:


stdin.getu16();
mov( ax, u16Var );

HLA low-level calling sequence examples:


call stdin.getu16;
mov( ax, u16Var );
```

stdin.getu32(); @returns("eax");

This function reads an unsigned 32-bit decimal integer in the range 0..+4,294,967,295 from the standard input. 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. This function allows underscores in the interior of the number. The stdin.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:


stdin.getu32();
mov( eax, u32Var );

HLA low-level calling sequence examples:
```

```
call stdin.getu32;
mov( eax, u32Var );
```

stdin.getu64( ); @returns("edx:eax");

This function reads an unsigned 64-bit decimal integer from the standard input. 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 stdin.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 register pair (EDX holds the H.O. dword).

HLA high-level calling sequence examples:

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

HLA low-level calling sequence examples:

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

stdin.getu128(var dest:lword);

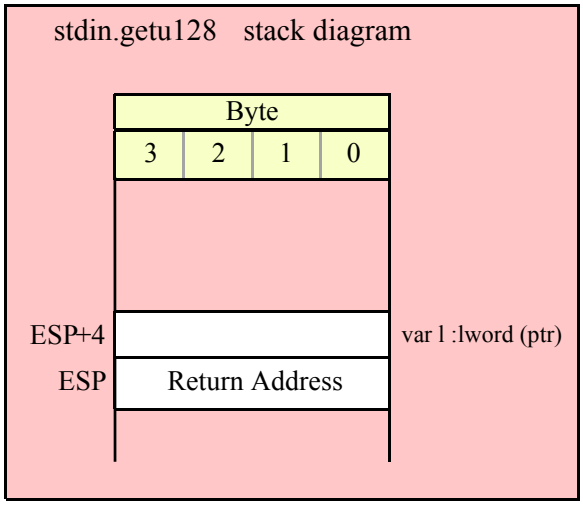
This function reads an unsigned 128-bit decimal integer from the standard input. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The stdin.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:

```
stdin.getu128( lwordVar );
```

HLA low-level calling sequence examples:

```
// If lwordVar is a static variable:

pushd( &lwordVar );
call stdin.getu128;

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

lea( eax, lwordVar ); // Assume EAX is available
push( eax);
call stdin.getu128;
```



28.9 Floating Point Input

`stdin.getf();`

This function reads an 80-bit floating point value in either decimal or scientific from from the standard input 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. 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:


stdin.getf();
fstp( fpVar );

HLA low-level calling sequence examples:


call stdin.getf;
fstp( fpVar );
```

28.10 Generic File Input

`stdin.get( List_of_items_to_read );`

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

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

The macro invocation above expands into the following:

```
push( eax );
stdin.geti32( i32 );
stdin.getc();
mov( al, charVar );
```

```
stdin.geti16();
mov( ax, u16 );
stdin.gets( strVar );
pop( eax );
```

Notice that stdin.get preserves the value in the EAX and EDX registers even though various stdin.get*xxx* functions use these registers. Note that stdin.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 stdin.get supports eight-bit, 16-bit, 32-bit, 64-bit, and 128-bit input values. It automatically selects the appropriate input routine based on the type of the variable you specify.