5 The Blobs Module (blobs.hhf)

This unit contains routines that read data from and write data to memory-based streams (blobs, or Binary Large OBjects). The blob functions can be broken down into five generic categories: general functions that initialize and allocate blobs, functions that convert between blobs and generic memory buffers, functions that do blob file I/O, accessor functions that provide access to the internal blob data structure, and blob I/O functions.

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

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

A note about thread safety: Each blob object maintains its own critical section variable. Therefore, blobs are protected from multi-threaded access on a per-blob level. Because blobs create this critical section object whenever a blob is initialized or allocated, you must ensure that you free or destroy the blob when you are done using it (and before your program quits) in order to free up system resources.

5.1 Conversion Format Control

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

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

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

5.2 Blob Synopsis

The best way to describe a blob (Binary Large Object) is by calling it a "memory-based file." You generally read and write data to a blob using a file I/O paradigm, that is, by using function calls similar to the HLA standard library's fileio functions. The advantage of blobs over files is that blobs are much faster than files; the disadvantage of blobs is that blob data is volatile (meaning that you lose all data written to a blob when the program quits or if there is a power outage). Another pair of blob disadvantages is that blobs' sizes are limited by the amount of available system RAM (files are limited by the amount of free disk space, which is usually much larger) and that you have to predeclare a blob's size before using it. By contrast, files can grow to any size up to the amount of free disk space you having to specify the maximum size prior to creating the file. Despite these limitations, blobs are quite useful because blob manipulation is much faster than file manipulation (often three orders of magnitude faster).

Blobs are very similar to strings. The main differences between strings and blobs are

- Strings generally hold character data (ASCII text), blobs hold arbitrary character and binary data (hence the name "Binary Large OBject").
- Strings generally represent a single coherent value (e.g., a single line of text), blobs may contain many records (e.g., multiple lines of text).
- Strings are generally created and manipulated in their entirety; blobs are generally built up and accessed using a sequence of operations (similar to file I/O operations).
- Blobs' data can be aligned on any address that is a power of two (and greater than or equal to 4); strings are always aligned on dword addresses. This makes blobs especially useful for manipulating data objects with certain SSE instructions.

5.3 Blob Internal Representation

The following is the internal representation of a blob object (these declarations are inside the blob namespace):

```
type
  blobRec:
     record := -20;

           allocPtr        :dword;
           criticalSection :dword
           rcursor         :dword;
           wcursor         :dword;
           maxlen          :dword;
           length          :dword;
           blobData        :byte[16];  // Minimum is 16 bytes
     endrecord;

  t         :pointer to blobRec;  // blob.t outside blob namespace
  blob_t    :t;                   // blob.blob_t outside blob namespace
  blob      :t;                   // blob.blob outside blob namespace
```

Note that the fieldnames in blobRec are generally considered private to the blobRec record declaration and are intended for internal use (by the blobs package) only. Applications should only reference or modify these fields using the blob accessor functions (described a little later in this document).

The allocPtr field contains the address of the start of the memory block allocated for the blob. Blobs can be allocated to any address (that is a power of two and greater than or equal to 4). Because the stdlib's mem.alloc function only guarantees 8-byte alignment, certain calls that allocate storage for a blob might need to allocate extra storage in order to align the blob's data (the blobData field) at an appropriate address in memory. This might insert (an arbitrary amount of) padding bytes before the allocPtr field in the blob data structure. The blob data structure needs to save this allocation address (in allocPtr) so that the application can free the storage consumed by the blob object. Much of the time, the allocPtr field will contain it's own address; for unaligned (the most common) blobs, or blobs aligned on 8-byte or less addresses, no extra padding is necessary. If the blob object is allocated in static memory rather than on the heap (or in memory that was not allocated by the blob library functions), the allocPtr field will contain NULL.

The criticalSection field contains the object that controls access to the blob's critical section in multi-threaded applications. Note that this field is still present (though unused) in single-threaded applications. This is a private field and should never be accessed by code outside the blob library.

The rcursor (read cursor) field contains the offset into the blob's data where the next value will be read from the blob using one of the blob.get* functions or any of the other blob input functions that read data from the read cursor position.

The wcursor (write cursor) field contains the offset into the blob's data where the next value will be written to in the blob using one of the blob.put* functions or any of the other blob output functions that write data to the write cursor position in the blob.

The maxlen (maximum length) field contains the maximum size of the blob. If an application attempts to read or write data beyond this point in the blob, the library routines will generate an exception.

The length field is the current (dynamic) length of the blob. This field's value is always less than or equal to maxlen's value.

The blobData field is where the blob's data appears in memory. This is an array of bytes containing at least maxlen bytes (it may contain more, depending on the alignment of the blob allocation). This field always contains at least 16 bytes (hence the declared size of 16 in the blobRec record).

5.4 Declaring Blob Variables

HLA stdlib blob variables are pointers. When declaring blob objects, you should use the blob.blob, blob.blob_t, or blob.t data types. These three names are all synonyms, you can use whatever form you choose. Whenever this documentation refers to a "blob variable", it is actually talking about a pointer to a blob object, that is, a variable of type blob.blob, blob.blob_t, or blob.t. For example:

```
var
  blobVar    : blob.t;
```

```
blobVar2    : blob.blob_t;
blobVar3    : blob.blob;
```

Note that these declarations all reserve exactly four bytes for the blob variables (blobVar1, blobVar2, and blobVar3). Blob variables always hold pointers to the actual blob data. Of course, like strings and other pointers, these declarations do not actually allocate storage for the blob object itself. You will have to call a function such as blob.alloc to allocate the actual storage for a blob object.

You can also use the blob.init function to initialize a block of memory you've allocated for use as a blob. See the description of this function for more details.

5.4.1 Initializing and Allocating Blob Variables

The HLA Blobs module provides several functions that let you allocate storage for a blob on the heap or associate other storage with the blob. Because you must allocate storage for a blob before using it, these functions will generally be the first functions you call before using a blob.

```
blob.init( var b:var; numBytes:dword ); @returns( "eax" );
```

The blob.init routine initializes a block of memory so that it contains a blob object. This function returns a pointer to the blob object in EAX, which you should store into a blob.t variable. The b argument is the address of a block of bytes that you want to use as the blob object. The numBytes argument is the total size of the block of memory pointed at by the b parameter. Note that at least 20 bytes of the object pointed at by b will be used to hold the blob's metadata (internal data structure). That is, numBytes does not specify the maximum size of the blob; the maximum size will actually be slightly smaller than this value. This function will raise an exception if numBytes is too small (insufficient space to allocate the blob metadata and at least 16 bytes of blob data.). **Warning**: that this function sets the allocPtr field of the blobRec data structure to NULL. If you have allocated this storage on the heap, it is your responsibility to call mem.free with the original allocation address to return the storage to the heap. You must never call blob.free to free the blob storage that you've initialized with a blob.init call.

Important: this function creates a new critical section object and initializes the criticalSection field of the blob data object.

You must call blob.destroy when you are done with the blob to delete the critical section object and return its resources to the operating system.

```
blob.init16( var b:var; numBytes:dword ); @returns( "eax" );
```

Just like blob.init except that this function always returns an address that is aligned to a 16-byte boundary. Note that if the memory block pointed at by b isn't at an appropriate address, blob.init16 will use up to 15 bytes at the beginning of this block as padding bytes to guarantee that the address this function returns in eax is aligned to a 16-byte boundary. Therefore, you should ensure that the block of memory whose address you pass is at least 16 bytes larger than you need to ensure you have enough space after padding is removed from the total. The warnings above apply to blob.init16 as well as blob.init.

```
blob.alloc( size:dword); @returns( "eax" );
```

The blob.alloc routine allocates storage for a blob object that contains at least size bytes of blob data. It initializes the blob header (blobRec) data structure and returns a pointer to the blob in EAX (that you should store into a blob variable). This function will raise an exception if it cannot allocate the specified amount of storage on the heap for the blob object. Note that this function also allocates storage for the blob header and pads the size of the blob (typically to a multiple of 16 bytes) so this call allocates a little bit more storage than size bytes on the heap. Note that blobs created with the blob.alloc function are always aligned on a 16-byte boundary and they always initialize the blobRec allocPtr field with the address of the storage allocated by an (internal) mem.alloc call.

Important: this function creates a new critical section object and initializes the criticalSection field of the blob data object.

You must call blob.free when you are done with the blob to delete the critical section object and return its resources to the operating system.

blob.allocAligned(size:dword; alignment:dword); @returns("eax");

The blob.allocAligned routine also allocates storage for a blob object on the heap. The alignment argument is an integer in the range 0..16 specifying the alignment on a 2**alignment byte boundary. That is, the alignment value is interpreted as follows:

alignment	Blob data aligned to
0	16 bytes
1	16 bytes
2	16 bytes
3	16 bytes
4	16 bytes
5	32 bytes
6	64 bytes
7	128 bytes
8	256 bytes
9	512 bytes
10	1024 bytes
11	2048 bytes
12	4096 bytes
13	8192 bytes
14	16384 bytes
15	32768 bytes
16	65536 bytes

Note that the minimum alignment is always 16 bytes because this is the alignment that blob.alloc guarantees. Note that the blob.allocAligned function might have to add as many as 2*alignment+32 bytes to the size of the storage allocated on the heap in order to guarantee alignment on the desired address boundary. As for the blob.alloc call, this function initializes the internal allocPtr field with the address of the block of storage allocated on the heap.

Important: this function creates a new critical section object and initializes the criticalSection field of the blob data object.

You must call blob.free when you are done with the blob to delete the critical section object and return its resources to the operating system.

blob.realloc(theBlob:blob.t; size:dword); @returns("eax");

The blob.realloc routine resizes an existing blob. The first argument, theBlob, is the address of the blob object to resize and the second argument is the new maximum size of the resultant blob. This function works by allocating a new blob of the specified size, copying the blob data from the original blob to the new blob, and then freeing the storage associated with the original blob if the allocPtr field contains a non-NULL value.

It will also destroy the critical section object held by the original blob.

Important: this function creates a new critical section object and initializes the criticalSection field of the blob data object.

You must call blob.free when you are done with the blob to delete the critical section object and return its resources to the operating system.

Note that there is no blob.reallocAligned function call, but you can easily write your own using the following code:

```
blob.allocAligned( newsize, desiredAlignment );
push( eax );
blob.cpy( originalBlob, eax );
blob.free( originalBlob );
```

```
    pop( eax );
    mov( eax, originalBlob );
```

blob.free(theBlob:blob.t);

The blob.free routine frees the storage associated with a blob object if that blob object was allocated on the heap with a call to blob.alloc or blob.allocAligned (specifically, if the allocPtr field in the blobRec data structure contains a non-NULL value). If the allocPtr field contains NULL, then this function returns without raising an exception. If theBlob is NULL, contains a bad address, or if allocPtr isn't pointing at an object on the heap, then this routine will raise an appropriate exception.

This function will also delete the critical section object held b the blob.

blob.destroy(theBlob:blob.t);

The blob.destroy routine deletes the critical section held by the blob. If the allocPtr

field is non-NULL, this will also deallocate the storage held by the blob. Functionally, blob.free and blob.destroy are equivalent; though blob.free is intended for blobs allocated via some blob.allooc* or blob.a_* function and blob.destroy is intended for blobs initialized via the blob.init* functions.

5.5 Blob Accessor Functions

The following functions in the HLA blobs unit provide access to the blobRec data structure. Programs should use these accessor functions rather than directly accessing the blobRect fields.

blob.length(b:blob.t); @returns("eax");

The blobl.length routine returns the value of the blobRec.length field, that is, the current size (actual data) of the blob.

```
HLA high-level calling sequence example:

blob.length( b );
  mov( eax, blobLength );


HLA low-level calling sequence example:

push( b );
call blob.length;
mov( eax, blobLength );
```

blob.setLength(b:blob.t; newLen:dword);

The blobl.setLength routine sets the value of the blobRec.length field to the value specified by the newLen parameter. You should exercise extreme caution when using this function. The blob.setLength function does not check the value of the newLen argument to determine if it is valid. You could supply a value larger than the actual amount of data currently in the blob (meaning you've just added garbage to the end of the blob) or you could even supply a value that is beyond the allocated size of the blob (creating memory access problems down the road).

```
HLA high-level calling sequence example:

blob.setLength( b, 32768 );

HLA low-level calling sequence example:

push( b );
pushd( 32768 );
```

```
  call blob.setLength;
```

blob.maxlen(b:blob.t); @returns("eax");

The blob.maxlen routine returns the value of the blobRec.maxlen field, that is, the current maximum size of the blob.

```
  HLA high-level calling sequence example:

  blob.maxlen( b );
    mov( eax, blobMaxLen );

  HLA low-level calling sequence example:

  push( b );
  call blob.maxlen;
  mov( eax, blobMaxLen );
```

blob.setMaxlen(b:blob.t; newLen:dword);

The blobl.setMaxlen routine sets the value of the blobRec.maxlen field to the value specified by the newLen parameter. You should exercise extreme caution when using this function. The blob.setMaxlen function does not check the value of the newLen argument to determine if it is valid. You should only set the value of this field when allocating storage for a new blob or if you are shrinking the size of a blob in-place. Expanding the maxlen value does not allocate any more storage for the blob and such action will probably lead to a memory fault later on in your program. Note that this function does not change the value of the blobRec.length field, even if the new value for maxlen is less than the existing length.

```
  HLA high-level calling sequence example:

  blob.setMaxlen( b, 32768 );


  HLA low-level calling sequence example:

  push( b );
  pushd( 32768 );
  call blob.setMaxlen;
```

blob.rcursor(b:blob.t); @returns("eax");

The blobl.rcursor routine returns the value of the blobRec.rcursor field in the EAX register.

```
  HLA high-level calling sequence example:

  blob.rcursor( b );
    mov( eax, blobReadPosition );

  HLA low-level calling sequence example:

  push( b );
  call blob.rcursor;
  mov( eax, blobReadPosition );
```

blob.setrCursor(b:blob.t; newCursor:dword);

The blobl.setrCursor routine sets the value of the blobRec.rcursor field to the value specified by the newCursor parameter. You should exercise extreme caution when using this function. The blob.setrCursor function does not check the value of the newCursor argument to determine if it is valid. If you point the rcursor field beyond the end of the blob's length you could get a memory fault error or read garbage data.

```
HLA high-level calling sequence example:

blob.setrCursor( b, 0 );

HLA low-level calling sequence example:

push( b );
pushd( 0 );
call blob.setrCursor;
```

blob.wcursor(b:blob.t); @returns("eax");

The blobl.wcursor routine returns the value of the blobRec.wcursor field in the EAX register.

```
HLA high-level calling sequence example:

blob.wcursor( b );
  mov( eax, blobWritePosition );

HLA low-level calling sequence example:

push( b );
call blob.wcursor;
mov( eax, blobWritePosition );
```

blob.setwCursor(b:blob.t; newCursor:dword);

The blobl.setwCursor routine sets the value of the blobRec.rcursor field to the value specified by the newCursor parameter. You should exercise extreme caution when using this function. The blob.setwCursor function does not check the value of the newCursor argument to determine if it is valid. If you point the wcursor field beyond the end of the blob's length you could get a memory fault error or leave garbage data in the middle of the blob.

```
HLA high-level calling sequence example:

blob.setwCursor( b, 0 );

HLA low-level calling sequence example:

push( b );
pushd( 0 );
call blob.setwCursor;
```

blob.reset;

The blobl.reset routine sets the blob's rcursor, wcursor, and length fields to zero. This effectively restores the blob to its original state when it was created. Note that this does not change any actual data appearing the the blob's memory storage, but since blob.reset sets the length field to zero, this effectively erases any data present in the blob.

```
HLA high-level calling sequence example:

blob.reset();

HLA low-level calling sequence example:

call blob.reset;
```

blob.eof(b:blob.t); @returns("@c");

This function returns true (1) in EAX and the carry flag if the read cursor is at the end of the blob's data. It returns false (0) otherwise. Note that this function actually returns true/false in EAX even though the "returns" value is "@c". It also returns the EOF state in the carry flag (c=1 if EOF, c=0 if not at EOF).

```
HLA high-level calling sequence example:

while( !(blob.eof( blobPointer )) do

     << something while not at EOF>>

  endwhile;

HLA low-level calling sequence example:

whileNotEOF:
  push( blobPointer );
  call blob.eof;
  cmp( al, true );
  jne atEOF;


     << something while not at EOF>>

  jmp whileNotEOF;
atEOF:
```

5.6 Blob Assignment Functions

These functions copy blobs and fill a blob with a single byte, word, or double-word value.

blob.a_cpy(b:blob.t); @returns("eax");

This function creates a new blob on the heap that is a copy of the blob pointed at by the b parameter. It returns a pointer to the new blob in the EAX register. When you are done using this new blob, you should free the storage associated with it (and delete the criticalsection it creates) by calling the blob.free function.

```
HLA high-level calling sequence example:

blob.a_cpy( someBlob );
mov( eax, newBlob );
```

```
HLA low-level calling sequence example:

push( someBlob );
call blob.a_cpy;
mov( eax, newBlob);
```

blob.cpy(src:blob.t; dest:blob.t); @returns("eax");

This function copies the data from the source blob (pointed at by src) to the destination blob (pointed at by dest) and returns a pointer to the destination blob in EAX. This function raises an ex.BlobOverflow exception if the destination blob isn't large enough to hold the data copied from the source blob.

```
HLA high-level calling sequence example:

blob.cpy( someBlob, destBlob );

HLA low-level calling sequence example:

push( someBlob );
push( destBlob );
call blob.cpy;
```

blob.fillb(theValue:byte; numBytes:dword; dest:blob);

This function fills an existing blob with numBytes copies of the byte value theValue. This function resets the rcursor field to zero and the wcursor and length fields to numBytes. This function raises an ex.BlobOverflow exception if the destination blob (pointed at by dest) is not large enough to hold numBytes bytes.

```
HLA high-level calling sequence example:

blob.fillb( 0, 1024, destBlob );

HLA low-level calling sequence example:

pushd( 0 );
pushd( 1024 );
push( destBlob );
call blob.fillb;
```

blob.fillw(theValue:word; numWords:dword; dest:blob);

This function fills an existing blob with numWords copies of the word value theValue. This function resets the rcursor field to zero and the wcursor and length fields to numWords*2. This function raises an ex.BlobOverflow exception if the destination blob (pointed at by dest) is not large enough to hold numWords*2 bytes.

```
HLA high-level calling sequence example:

blob.fillw( 1000, 1024, destBlob );

HLA low-level calling sequence example:

pushd( 1000 );
pushd( 1024 );
push( destBlob );
call blob.fillw;
```

blob.filld(theValue:word; numDwords:dword; dest:blob);

This function fills an existing blob with numDwords copies of the dword value theValue. This function resets the rcursor field to zero and the wcursor and length fields to numDwords*4. This function raises an ex.BlobOverflow exception if the destination blob (pointed at by dest) is not large enough to hold numDwords*4 bytes.

```
HLA high-level calling sequence example:

blob.filld( 1_000_000, 1024, destBlob );

HLA low-level calling sequence example:

pushd( 1_000_000 );
pushd( 1024 );
push( destBlob );
call blob.filld;
```

5.7 Blob Extraction Functions

These funcions extract subranges (slices) of a blob.

blob.a_subBlob(src:blob; start:dword; len:dword); @returns("eax");

This function creates a new blob on the heap and initializes it with data from an existing blob (pointed at by src). The new blob is initialized with len bytes starting at offset start in blob src. This function returns a pointer to the new blob on the heap in the EAX register. It is the callers responsibility to call blob.free to free the storage associated with this new blob (and delete the critical section that this function creates).

```
HLA high-level calling sequence example:

blob.a_subBlob( someBlob, 1024, 256 );
mov( eax, newBlob );

HLA low-level calling sequence example:

push( someBlob );
pushd( 1024 );
pushd( 256 );
call blob.a_subBlob;
mov( eax, newBlob );
```

blob.subBlob(src:blob; start:dword; len:dword; dest:blob.t);

This function copies a sequence from a source blob (src) to a destination blob (dest). It copies len bytes starting at offset start in src to dest. This function raises an ex.BlobOverflow exception if d is too small to receive the blob data.

```
HLA high-level calling sequence example:

blob.subBlob( someBlob, 1024, 256, destBlob );

HLA low-level calling sequence example:

push( someBlob );
pushd( 1024 );
pushd( 256 );
push( destBlob
call blob.subBlob;
```

5.8 Blob Comparison Functions

These funcions compare two blobs for equality or inequality.

blob.eq(left:blob; right:blob.t); @returns("@c");

This function compares two blobs and returns true in the carry flag and the AL register if the two blobs are equal to one another.

```
HLA high-level calling sequence example:

if( blob.eq( someBlob, anotherBlob ) ) then

     // Do something if blobs are equal

endif;

HLA low-level calling sequence example:

push( someBlob );
push( anotherBlob );
call blob.eq;
jnc blobsNotEqual;

     // Do something if blobs are equal

blobsNotEqual:
```

blob.ne(left:blob; right:blob.t); @returns("@c");

This function compares two blobs and returns true in the carry flag and the AL register if the two blobs are not equal to one another.

```
HLA high-level calling sequence example:

if( blob.ne( someBlob, anotherBlob ) ) then

     // Do something if blobs are not equal

endif;

HLA low-level calling sequence example:

push( someBlob );
push( anotherBlob );
call blob.eq;
jnc blobsEqual;

     // Do something if blobs are not equal

blobsEqual:
```

5.9 Blob Scanning Functions

These funcions convert memory buffers (ranges of bytes) to blob objects

```
blob.index( src1:blob; src2:blob.t ); @returns( "@c" );

blob.index( src1:blob; offs:dword src2:blob.t ); @returns( "@c" );

blob.index2( src1:blob; src2:blob.t ); @returns( "@c" );

blob.index3( src1:blob; offs:dword src2:blob.t ); @returns( "@c" );

blob.indexStr( src1:blob; src2:string ); @returns( "@c" );

blob.indexStr( src1:blob; offs:dword src2:string ); @returns( "@c" );

blob.indexStr2( src1:blob; src2:string ); @returns( "@c" );

blob.indexStr3( src1:blob; offs:dword src2:string ); @returns( "@c" );
```

These functions search for the presence of a string or blob within some blob. They return the carry flag set if they find the src2 blob or string within the src1 blob and clear if the src2 string or blob is not found within the src1 blob. The variants with the offs parameter begin searching for the blob or string at offset offs within the blob src1. If the carry comes back set (meaning src2 was found), then the EAX register will contain the offset into src1 where src1 is first found. These functions raise an ex.ValueOutOfRange exception if the offs value is greater than the current length of src1.

```
HLA high-level calling sequence example:

if( blob.index( someBlob, subBlob ) ) then

     // Do something if subBlob is a subblob of someBlob

endif;

HLA low-level calling sequence example:

push( someBlob );
push( subBlob );
call blob.index;
jnc notPresent;

     // Do something if subBlob is a subblob of someBlob

notPresent:
```

```
blob.rindex( src1:blob; src2:blob.t ); @returns( "@c" );

blob.rindex( src1:blob; offs:dword src2:blob.t ); @returns( "@c" );

blob.rindex2( src1:blob; src2:blob.t ); @returns( "@c" );

blob.rindex3( src1:blob; offs:dword src2:blob.t ); @returns( "@c" );

blob.rindexStr( src1:blob; src2:string ); @returns( "@c" );

blob.rindexStr( src1:blob; offs:dword src2:string ); @returns( "@c" );

blob.rindexStr2( src1:blob; src2:string ); @returns( "@c" );

blob.rindexStr3( src1:blob; offs:dword src2:string ); @returns( "@c" );
```

These functions search backwards for the presence of a string or blob within some blob starting at the end of that blob. They return the carry flag set if they find the src2 blob or string within the src1 blob and clear if the src2 string or blob is not found within the src1 blob. The variants with the offs parameter begin searching for the blob or string at offset length-offs within the blob src1. If the carry comes back set (meaning src2 was found), then the

EAX register will contain the offset into src1 where src1 is last found. These functions raise an ex.ValueOutOfRange exception if the offs value is greater than the current length of src1.

```
HLA high-level calling sequence example:

if( blob.rindex( someBlob, subBlob ) ) then

     // Do something if subBlob is a subblob of someBlob

endif;

HLA low-level calling sequence example:

push( someBlob );
push( subBlob );
call blob.rindex;
jnc notPresent;

     // Do something if subBlob is a subblob of someBlob

notPresent:
```

blob.chpos(src1:blob; src2:char); @returns("@c");

blob.chpos(src1:blob; offs:dword src2:char); @returns("@c");

blob.chpos2(src1:blob; src2:char); @returns("@c");

blob.chpos3(src1:blob; offs:dword src2:char); @returns("@c");

These functions search for the presence of a character within some blob. They return the carry flag set if they find the src2 character within the src1 blob and clear if the src2 character is not found within the src1 blob. The variants with the offs parameter begin searching for the character at offset offs within the blob src1. If the carry comes back set (meaning src2 was found), then the EAX register will contain the offset into src1 where src2 is first found. These functions raise an ex.ValueOutOfRange exception if the offs value is greater than the current length of src1.

```
HLA high-level calling sequence example:

if( blob.chpos( someBlob, someChar ) ) then

     // Do something if someChar is within someBlob

endif;

HLA low-level calling sequence example:

push( someBlob );
mov( someChar, al );
push( eax );
call blob.chpos;
jnc notPresent;

     // Do something if someChar is within someBlob

notPresent:
```

```
blob.rchpos( src1:blob; src2:char ); @returns( "@c" );

blob.rchpos( src1:blob; offs:dword src2:char ); @returns( "@c" );

blob.rchpos2( src1:blob; src2:char ); @returns( "@c" );

blob.rchpos3( src1:blob; offs:dword src2:char ); @returns( "@c" );
```

These functions search for the presence of a character within some blob, searching from the end of the blob. They return the carry flag set if they find the src2 character within the src1 blob and clear if the src2 character is not found within the src1 blob. The variants with the offs parameter begin searching for the character at offset offs within the blob src1. If the carry comes back set (meaning src2 was found), then the EAX register will contain the offset into src1 where src2 is first found. These functions raise an ex.ValueOutOfRange exception if the offs value is greater than the current length of src1.

```
HLA high-level calling sequence example:

if( blob.rchpos( someBlob, someChar ) ) then

     // Do something if someChar is within someBlob

endif;

HLA low-level calling sequence example:

push( someBlob );
mov( someChar, al );
push( eax );
call blob.rchpos;
jnc notPresent;

     // Do something if someChar is within someBlob

notPresent:
```

5.10 Blob Concatenation Functions

There are two major types of blob concatentation functions. The first group (consisting of the blob.cat macro and the blob.cat2, blob.cat3, and blob.a_cat functions) take all the data from one blob and concatenates that data to the end (that is, at the length offset) of a second blob. These functions set the length and wcursor fields to point at the end of the new blob and reset the rcursor position of the result to zero.

The second group of concatenation functions take data from a string, a buffer, or a blob and append this to the end of some destination blob starting at the wcursor position in the destination blob.

```
blob.a_cat( src1:blob; src2:char ); @returns( "@eax" );
```

This function creates a new blob on the heap whose size is equivalent to the current lengths of the src1 and src2 blobs whose pointers are passed as parameters. This function then copies the data from src1 to the new blob and then appends the data from src2 to the end of this. This function returns a pointer to the new blob in EAX and the length and wcursor fields are set to the new blob's length and the rcursor field is set to zero. Note that it is the caller's responsibility to call blob.free in order to return the allocated storage to the heap and delete the newly created critical section object for the blob.

```
HLA high-level calling sequence example:

blob.a_cat( blob1, blob2 );
mov( eax, newBlob );

HLA low-level calling sequence example:
```

```
    push( blob1 );
    push( blob2 );
    call blob.a_cat;
    mov( eax, newBlob );
```

blob.cat(src:blob; dest:blob);

blob.cat2(src:blob; dest:blob);

These functions concatenate the data from the src blob to the end of the data in the dest blob. They raise an ex.BlobOverflow exception if the result will not fit in the destination blob.

```
    HLA high-level calling sequence example:

    blob.cat2( srcblob, destblob );

    HLA low-level calling sequence example:

    push( srcblob );
    push( destblob );
    call blob.cat2;
    mov( eax, newBlob );
```

blob.cat(src1:blob; src2:blob; dest:blob);

blob.cat3(src1:blob; src2:blob; dest:blob);

These functions concatenate the data from two blobs (src1 and src2) and store the concatenated result into the dest operand. They raise an ex.BlobOverflow exception if the result will not fit in the destination blob.

```
    HLA high-level calling sequence example:

    blob.cat3( blob1, blob2, dest );

    HLA low-level calling sequence example:

    push( blob1 );
    push( blob2 );
    push( dest );
    call blob.cat3;
```

blob.catsub(src:blob; start:dword; len:dword; dest:blob);

blob.catsub4(src:string; start:dword; len:dword; dest:blob);

blob.catsub(src2:blob; start:dword; len:dword; src1:string; dest:blob);

blob.catsub5(src2:string; start:dword; len:dword; src1:string; dest:blob);

The blob.catsub functions are actually overloaded procedures that map to the blob.catsub4 and blob.catsub5 functions, depending on the call signature.

These functions concatenate the data froma string (src) or pair of strings (src1 and src2) and store the concatenated result into the dest blob operand. They raise an ex.BlobOverflow exception if the result will not fit in the destination blob. These functions raise an ex.StringIndexError exception if the start index value is greater than the current length of the string.

The first two functions extract a substring of length len, starting at character position start, from src and concatenate this string to the end of the dest blob.

The second two functions copy the src1 string to the blob and then copy the substring of src2 (specified by start and len) to the end of this blob.

All of these functions concatenate their strings to the blob starting at the wcursor position in the blob. They will leave wcursor pointing beyond the data just concatenated and will update the length field of the blob if the concatenated data extends the length. These functions do not affect the rcursor field of the blob.

```
HLA high-level calling sequence example:

blob.catsub4( strVar, 0, 24, dest );
blob.catsub5( strVar2, 0, 24, strVar1, dest );

HLA low-level calling sequence example:

push( strVar );
pushd( 0 );
pushd( 24 );
push( dest );
call blob.catsub4;

push( strVar2 );
pushd( 0 );
pushd( 24 );
push( strVar1 );
push( dest );
call blob.catsub5;
```

blob.a_catsub(src:blob; start:dword; len:dword; dest:blob);

This function extracts a substring (src, from start of length len) and creates a new blob on the heap from the substring data. This function returns a pointer to the new blob in EAX This function sets the rcursor field of the blob to zero and the wcursor and length fields of the blob to len. It is the caller's responsibility to free the storage allocated by the function (and the critical section it creates) by calling blob.free when the caller is done with this blob.

```
HLA high-level calling sequence example:

blob.a_catsub( strVar, 0, 24 );
mov( eax, newBlob );

HLA low-level calling sequence example:

push( strVar );
pushd( 0 );
pushd( 24 );
call blob.a_catsub;
mov( eax, newBlob );
```

blob.catbuf2(src:buf_t; dest:blob);

blob.catbuf3a(startBuf:dword; endBuf:dword; dest:blob);

blob.catbuf2 is a synonym for blob.catbuf3a. As it turns out, a buf_t object passed on the stack is the same as passing the startBuf and endBuf dwords on the stack.

This function concatenates the bytes from a buffer (specified by the src or startBuf/endBuf variables) to the end of an existing blob (dest). This function stores the buffer data starting at the offset specified by the wcursor field of the blob. The bytes from memory locations startBuf to endBuf-1 are concatenated to the blob. If the new wcursor field value is greater than length, this function also extends the value of the length field. This function raises an ex.BlobOverflow exception if the new blob size would be greater than the maxlen field value.

```
HLA high-level calling sequence example:

blob.catbuf3a( startAddress, endAddressPlus1, destBlob );
```

```
HLA low-level calling sequence example:

push( startAddress );
push( endAddressPlus1 );
pushd( destBlob );
call blob.catbuf3a;
```

blob.catbuf3b(src2:buf_t; src1:string; dest:blob);

blob.catbuf4(startBuf:dword; endBuf:dword; strSrc:string; dest:blob);

blob.catbufe3b is a synonym for blob.catbuf4. As it turns out, a buf_t object passed on the stack is the same as passing the startBuf and endBuf dwords on the stack.

This function concatenates the bytes from a buffer (specified by the src2 or startBuf/endBuf variables) to the end of a string (strSrc) and concatenate this data to the end of an existing blob (dest). This function stores the buffer data starting at the offset specified by the wcursor field of the blob. The bytes from the string (strSrc) and then memory locations startBuf to endBuf-1 are concatenated to the blob. If the new wcursor field value is greater than length, this function also extends the value of the length field. This function raises an ex.BlobOverflow exception if the new blob size would be greater than the maxlen field value.

```
HLA high-level calling sequence example:

blob.catbuf4( startAddress, endAddressPlus1, someStr, destBlob );

HLA low-level calling sequence example:

push( startAddress );
push( endAddressPlus1 );
push( someStr );
pushd( destBlob );
call blob.catbuf4;
```

5.11 Blob Conversion Functions

These funcions convert memory buffers (ranges of bytes) to blob objects

and strings to blob objects.

blob.bufToBlob2(buf:@global:buf_t; b:blob.t);

blob.bufToBlob3(startBuf:dword; endBuf:dword; b:blob.t);

These functions convert a

range of bytes (specified by a starting and ending address) into a blob object. The blob data is stored into the (previously allocated) blob object pointed at by the b parameter.

Note that these function names are actually synonyms for the same function. As it turns out, passing a buf_t object on the stack produces the exact same stack frame as passing a starting and ending buffer address.

The startBuf parameter is the address of the first byte of the memory block to convert; the endBuf parameter supplies the last address of the buffer *plus one*.

The b parameter must point at a previously allocated and initialized blob object. This blob must be large enough (maxlen) to hold the range of bytes specified by the buffer parameter(s).

```
HLA low-level calling sequence examples:

blob.bufToBlob3( startingAddress, endingAddress, blobPtr1 );
blob.bufToBlob3( ebx, ecx, blobPtr2 );
blob.bufToBlob2( buf_t_Variable, blobPtr3 );

HLA low-level calling sequence examples:
```

```
    push( startingAddress );
    push( endingAddress );
    push( blobPtr1 );
    call blob.bufToBlob3;

    push( ebx );
    push( ecx );
    push( blobPtr2 );
    call blob.a_bufToBlob2;

    push( (type dword buf_t_Variable[0]) );
    push( (type dword buf_t_Variable[4]) );
    push( blobPtr3 );
    call blob.a_bufToBlob1;
```

blob.a_bufToBlob1(buf:@global:buf_t); @returns("@eax");

blob.a_bufToBlob2(startBuf:dword; endBuf:dword); @returns("@eax");

These functions convert a range of bytes (specified by a starting and ending address) into a blob object. The blob's data is allocated on the heap and these functions return a pointer to the blob data in the EAX register.

Note that these function names are actually synonyms for the same function. As it turns out, passing a buf_t object on the stack produces the exact same stack frame as passing a starting and ending buffer address.

The startBuf parameter is the address of the first byte of the memory block to convert; the endBuf parameter supplies the last address of the buffer *plus one*.

It is the caller's responsibility to call blob.free to free up the allocated storage and release the critical section object when the caller is done using the blob these function create.

```
    HLA low-level calling sequence examples:

    blob.a_bufToBlob2( startingAddress, endingAddress );
    mov( eax, blobPtr1);
    blob.a_bufToBlob2( ebx, ecx );
    mov( eax, blobPtr2);
    blob.a_bufToBlob1( buf_t_Variable );
    mov( eax, blobPtr3);

    HLA low-level calling sequence examples:

    push( startingAddress );
    push( endingAddress );
    call blob.a_bufToBlob2;
    mov( eax, blobPtr1);

    push( ebx );
    push( ecx );
    call blob.a_bufToBlob2;
    mov( eax, blobPtr2);

    push( (type dword buf_t_Variable[0]) );
    push( (type dword buf_t_Variable[4]) );
    call blob.a_bufToBlob1;
    mov( eax, blobPtr3);
```

```
blob.strToBlob( src:string; dest:blob );

blob.zstrToBlob( src:string; dest:blob );
```

5.12 General Blob I/O Functions

Here are the blob file I/O routines provided by the HLA blobs unit:

`blob.a_load( FileName: string ); @returns( "eax" );`

The blobl.a_load routine opens the file by the specified name, allocates sufficient storage to hold all the data in the file, reads the file's data into the blob, and then closes the file. This function returns a pointer to the initialized blob in the EAX register. You should call blob.free to return this storage to the heap when you are done using the blob.

This function initializes the read cursor so that it points at the beginning of the blob data read from the file. It initializes the write cursor to point at the end of the blob's data; note, however, that there is no additional space allocated at the end of the blob, so any attempt to write data to the blob (without resetting the write cursor to some other point in the blob) will produce a blob overflow exception.

`blob.a_loadExtended( FileName: string; extend:dword ); @returns( "eax" );`

The blobl.a_loadExtended routine opens the file by the specified name, allocates sufficient storage to hold all the data in the file plus the number of bytes specified by the extend argument, reads the file's data into the blob, and then closes the file. This function returns a pointer to the initialized blob in the EAX register. You should call blob.free to return this storage to the heap when you are done using the blob.

This function initializes the read cursor so that it points at the beginning of the blob data read from the file. It initializes the write cursor to point at the end of the blob's data. Because the blob's size has been extended by the value of the second parameter in the call, you can write that many additional bytes to the file.

`blob.load( filename:string; b:blob.t );`

The blob.load routine opens the file by the specified name and reads the file's data into the blob specified by the b parameter. This routine raise an exception if there is a problem opening the file (e.g., the file does not exist). If the file is successfully opened, this function will read the file's data into the blob (raising an exception if the file's data is too large to fit in the blob or if there is an error reading the file).

```
HLA high-level calling sequence examples:

blob.load( filenameStr, b );
  blob.a_load( "myfile2.txt" );
  mov( eax, b2 );
  blob.a_loadExtended( "myfile3.txt", 8192 );
  mov( eax, b3 );

HLA low-level calling sequence examples:

push( filenameStr );
push( b );
call blob.open;

// Note: If you want to use a string literal for the filename, the best
// solution is to create a string object in the readonly section, e.g.,
//
// readonly
//   filenameStr2 :string := "myfile2.txt";
//
// and just use the "filenameStr2" object you've created.

push( filenameStr2 );
```

```
call blob.a_open;
mov( eax, b2 );

// You may also do the following if you have a register available:

lea( eax, "myfile3.txt" );
push( eax );
pushd( 8192 );
call blob.a_loadExtended;
mov( eax, b3 );
```

blob.appendFile(filename:string; b:blob.t.blob);

This function opens a file, reads its data, and appends that data to the end of an existing blob. It raises the ex.BlobOverflow exception if the file is to large to append to the end of the blob specified by the b parameter. This call sets the write cursor to the end of the file appended to the blob in memory; it does not affect the value of the read cursor.

```
HLA high-level calling sequence example:

blob.appendFile( fileNameStr, b );


HLA low-level calling sequence example:


push( filenameStr );
push( b );
call blob.appendFile;
```

blob.a_appendFile(filename:string; b:blob.t.blob); @returns("eax");

This function creates a new blob on the heap that is the size of the data in the b blob plus the size of the file specified by filename. It copies the data from the b blob to the new blob and then reads the file's data and appends that data to the end of the new blob. It returns a pointer to the new blob in the EAX register (the caller should ultimately call blob.free to return this storage to the heap). This call sets the write cursor to the end of the file appended to the blob in memory; the value of the read cursor will be the same value found in the b blob.

```
HLA high-level calling sequence example:

blob.a_appendFile( fileNameStr, b );
mov( eax, b2 );


HLA low-level calling sequence example:


push( filenameStr );
push( b );
call blob.a_appendFile;
mov( eax, b2 );
```

blob.a_appendFileExtended(filename:string; b:blob.t.blob; extend:dword)
{@returns("eax")};

This function allocates storage for a new blob that is the size of the existing blob plus the size of the file and the extend value. It then copies the blob specified by the b parameter to the newly allocated blob and appends the file's data to the end of this blob. Finally, it returns a pointer to the new blob in the EAX register. Note that the original blob (specified by the b parameter) is unaffected by this operation. This call sets the write cursor to the end of the file appended to the blob in memory; it sets the value of the read cursor to the same value of the original blob (passed in b).

```
HLA high-level calling sequence example:

blob.a_appendFileExtend( fileNameStr, b, 16384 );


HLA low-level calling sequence example:


push( filenameStr );
push( b );
pushd( 16384 )
call blob.a_appendFileExtend;
```

blob.save(filename:string; b:blob.t);

This function writes the blob's data (specified by the b parameter) to the file specified by the filename parameter. This function will overwrite any existing file.

```
HLA high-level calling sequence examples:

blob.save( fileNameStr, blobVar );


HLA low-level calling sequence examples:

push( fileNameStr );
push( blobVar );
call blob.save;
```

5.13 Blob Binary I/O Routines

blob.write(b:blob.t; var src:var; len:dword); @returns("eax");

This procedure writes the number of bytes specified by the len variable to the blob specified by the b parameter (at offset wcursor in the blob). The bytes starting at the address of the src object are written to the blob. No range checking is done on the src address value. It is your responsibility to ensure that the buffer contains at least len valid data bytes. Note that src is an untyped reference parameter. This means that blob.write will take the address of whatever object you provide as this parameter (including pointer variables, which may not be what you want). If you want to pass the value of a pointer variable as the buffer address (rather than the address of the pointer variable) when using the high-level style calling syntax, use the VAL keyword as a prefix to the parameter (see the following examples). This function returns the number of bytes written to the blob in the EAX register. If this operation writes bytes beyond the previous length of the blob, it will increment the blobRec.length field of the blob appropriately.

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

```
blob.write( blobPointer, buffer, count );

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

blob.write( blobPointer, val bufPtr, count );

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

blob.write( blobPointer, bufPtr, 4 );


HLA low-level calling sequence examples:

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

push( blobPointer );
pushd( &buffer );
push( count );
call blob.write;

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

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

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

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

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

push( blobPointer );
push( bufPtr);
push( count );
call blob.write;

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

push( blobPointer );
lea( eax, bufPtr );
push( eax );
```

```
pushd( 4 );
call blob.write;
```

blob.writeAt(b:blob.t; var src:var; index:dword; len:dword);
@returns("eax");

This procedure writes the number of bytes specified by the len variable to the blob specified by the b parameter at the offset specified by the index parameter. This procedure does not use nor does it modify the blobRec.wcursor value. The bytes starting at the address of the src object are written to the blob. No range checking is done on the src address value. It is your responsibility to ensure that the buffer contains at least len valid data bytes. Note that src is an untyped reference parameter. This means that blob.writeAt will take the address of whatever object you provide as this parameter (including pointer variables, which may not be what you want). If you want to pass the value of a pointer variable as the buffer address (rather than the address of the pointer variable) when using the high-level style calling syntax, use the VAL keyword as a prefix to the parameter (see the following examples). This function returns the number of bytes written to the blob in the EAX register. If the sum of index+len is greater than the previous length of the blob, then this function will extend the length of the blob. If the value of index is greater than the original length of the blob, then this function will return zero in EAX and will not transfer any data to the blob.

```
HLA high-level calling sequence example:

blob.writeAt( blobPointer, buffer, writeOffset, count );


HLA low-level calling sequence example:

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

push( blobPointer );
pushd( &buffer );
push( writeOffset );
push( count );
call blob.writeAt;
```

blob.putByte(b:blob.t; byteVal:byte);

This procedure writes a single byte value (byteVal, which is a single-byte binary value) to the blob specified by the b parameter (at offset wcursor in the blob). This function call advances the value of wcursor by 1. This call is effectively equivalent to blob.write(b, byteVal, 1); except that it does not return the number of bytes written in EAX (which is always 1, assuming there are no exceptions).

```
HLA high-level calling sequence example:

blob.putByte( blobPointer, ByteValue );


HLA low-level calling sequence example:

// Assumes buffer is a static object at a fixed
// address in memory and EAX is available for use:

push( blobPointer );
movzx( ByteValue, eax );
```

```
push( eax );
call blob.putByte;
```

blob.putWord(b:blob.t; wordVal:word);

This procedure writes a single word value (wordVal, which is a two-byte binary value) to the blob specified by the b parameter (at offset wcursor in the blob). This function call advances the value of wcursor by 2. This call is effectively equivalent to blob.write(b, wordVal, 2); except that it does not return the number of bytes written in EAX (which is always 2, assuming there are no exceptions).

```
HLA high-level calling sequence example:

blob.putWord( blobPointer, WordValue );


HLA low-level calling sequence example:

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

push( blobPointer );
pushw( 0 );
push( WordValue );
call blob.putWord;
```

blob.putDword(b:blob.t; dwordVal:dword);

This procedure writes a single dword value (dwordVal, which is a four-byte binary value) to the blob specified by the b parameter (at offset wcursor in the blob). This function call advances the value of wcursor by 4. This call is effectively equivalent to blob.write(b, dwordVal, 4); except that it does not return the number of bytes written in EAX (which is always 4, assuming there are no exceptions).

```
HLA high-level calling sequence example:

blob.putDword( blobPointer, DwordValue );


HLA low-level calling sequence example:

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

push( blobPointer );
push( DwordValue );
call blob.putDword;
```

blob.putQword(b:blob.t; QwordVal:qword);

This procedure writes a single qword value (qwordVal, which is an eight-byte binary value) to the blob specified by the b parameter (at offset wcursor in the blob). This function call advances the value of wcursor by 8. This call is effectively equivalent to blob.write(b, qwordVal, 8); except that it does not return the number of bytes written in EAX (which is always 8, assuming there are no exceptions).

```
HLA high-level calling sequence example:

blob.putQword( blobPointer, QwordValue );
```

```
HLA low-level calling sequence example:

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

push( blobPointer );
push( (type dword QwordValue[4]) );
push( (type dword QwordValue[0]) );
call blob.putQword;
```

blob.putTbyte(b:blob.t; tbyteVal:tbyte);

This procedure writes a single tbyte value (tbyteVal, which is a 10-byte binary value) to the blob specified by the b parameter (at offset wcursor in the blob). This function call advances the value of wcursor by 10. This call is effectively equivalent to blob.write(b, tbyteVal, 10); except that it does not return the number of bytes written in EAX (which is always 10, assuming there are no exceptions).

```
HLA high-level calling sequence example:

blob.putTbyte( blobPointer, TByteValue );


HLA low-level calling sequence example:

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

push( blobPointer );
pushw( 0 );
push( (type word  TByteValue [8]) );
push( (type dword TByteValue [4]) );
push( (type dword TByteValue [0]) );
call blob.putTbyte;
```

blob.putLword(b:blob.t; LwordVal:lword);

This procedure writes a single lword value (lwordVal, which is a 16-byte binary value) to the blob specified by the b parameter (at offset wcursor in the blob). This function call advances the value of wcursor by 16. This call is effectively equivalent to blob.write(b, lwordVal, 16); except that it does not return the number of bytes written in EAX (which is always 16, assuming there are no exceptions).

```
HLA high-level calling sequence example:

blob.putLword( blobPointer, LwordValue );


HLA low-level calling sequence example:

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

push( blobPointer );
push( (type dword LwordValue[12]) );
push( (type dword LwordValue[8 ]) );
push( (type dword LwordValue[4 ]) );
push( (type dword LwordValue[0 ]) );
```

```
call blob.putLword;
```

blob.read(b:blob.t; var buffer:byte; count:uns32); @returns("eax");

This routine reads a sequence of count bytes from the specified blob starting at the rcursor position in the blob. It stores the bytes into memory at the address specified by buffer.

It returns the number of bytes actually read from the blob in the EAX register (this is usually equal to the count value, unless the read operation attempts to read beyond the current length of the blob, in which case the actual number of bytes is returned in EAX).

```
HLA high-level calling sequence examples:


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


HLA low-level calling sequence examples:

// If buffer is a static variable:

push( blobPointer );
pushd( &buffer );
push( count );
call blob.read;
```

blob.readAt(b:blob.t; var buffer:byte; index:dword; len:uns32)

This routine reads a sequence of len bytes from the specified blob starting at offset index into the blob. It stores the bytes into memory at the address specified by buffer.

This function call ignores the initial value in the rcursor variable and it does not change this value. This function returns the actual number of bytes read in the EAX register (which is usually equal to len). If len plus index is greater than the current blob length, this this function returns the actual number of bytes read (which will be less than len) in the EAX register.

```
HLA high-level calling sequence examples:


blob.readAt( blobPointer, buffer, index, count );
blob.readAt( blobPointer, [eax], 500, 1024 );


HLA low-level calling sequence examples:


push( blobPointer );
pushd( &buffer );
push( index );
push( count );
call blob.readAt;
```

blob.getByte(b:blob.t); @returns("al");

This procedure reads a single byte value from the blob specified by the b parameter (at offset rcursor in the blob) and returns this byte in the AL register. This function call advances the value of rcursor by 1. This call is effectively equivalent to blob.read(b, byteVal, 1); except that it does not return the number of bytes read in EAX (which is always 1, assuming there are no exceptions).

It will raise an ex.BlobOverflow exception if the value of rcursor is greater than or equal to the current blob length.

```
HLA high-level calling sequence example:

blob.getByte( blobPointer );
mov( al, ByteValue );


HLA low-level calling sequence example:

push( blobPointer );
call blob.getByte;
mov( al, ByteValue );
```

blob.getWord(b:blob.t); @returns("ax");

This procedure reads a single word value from the blob specified by the b parameter (at offset rcursor in the blob) and returns this word in the AX register. This function call advances the value of rcursor by 2. This call is effectively equivalent to blob.read(b, wordVal, 2); except that it does not return the number of bytes read in EAX (which is always 2, assuming there are no exceptions).

It will raise an ex.BlobOverflow exception if the value of rcursor is greater than or equal to the current blob length.

```
HLA high-level calling sequence example:

blob.getWord( blobPointer );
mov( ax, WordValue );


HLA low-level calling sequence example:

push( blobPointer );
call blob.getWord;
mov( ax, WordValue );
```

blob.getDword(b:blob.t); @returns("eax");

This procedure reads a single dword value from the blob specified by the b parameter (at offset rcursor in the blob) and returns this dword in the EAX register. This function call advances the value of rcursor by 4. This call is effectively equivalent to blob.read(b, dwordVal, 4); except that it does not return the number of bytes read in EAX (which is always 4, assuming there are no exceptions).

It will raise an ex.BlobOverflow exception if the value of rcursor is greater than or equal to the current blob length.

```
HLA high-level calling sequence example:

blob.getDword( blobPointer );
mov( eax, DwordValue );


HLA low-level calling sequence example:

push( blobPointer );
call blob.getDword;
mov( ax, DwordValue );
```

blob.getQword(b:blob.t); @returns("edx:eax");

This procedure reads a single qword value from the blob specified by the b parameter (at offset rcursor in the blob) and returns this qword in the EDX:EAX register pair. This function call advances the value of rcursor by 8. This call is effectively equivalent to blob.read(b, qwordVal, 8); except that it does not return the number of bytes read in EAX (which is always 8, assuming there are no exceptions).

It will raise an ex.BlobOverflow exception if the value of rcursor is greater than or equal to the current blob length.

```
HLA high-level calling sequence example:

blob.getQword( blobPointer );
mov( eax, (type dword QwordValue[0]) );
mov( edx, (type dword QwordValue[4]) );


HLA low-level calling sequence example:

push( blobPointer );
call blob.getQword;
mov( eax, (type dword QwordValue[0]) );
mov( edx, (type dword QwordValue[4]) );
```

blob.getTbyte(b:blob.t; tbyteVal:tbyte);

This procedure reads a single tbyte value from the blob specified by the b parameter (at offset rcursor in the blob) and stores this tbyte via the tbyteVal reference parameter. This function call advances the value of rcursor by 10. This call is effectively equivalent to blob.read(b, tbyteVal, 10); except that it does not return the number of bytes read in EAX (which is always 10, assuming there are no exceptions).

It will raise an ex.BlobOverflow exception if the value of rcursor is greater than or equal to the current blob length.

```
HLA high-level calling sequence example:

blob.getTbyte( blobPointer, tByteVar );


HLA low-level calling sequence example:

push( blobPointer );
pushd( &tbyteVar );
call blob.getTbyte;
```

blob.getLword(b:blob.t; lwordVal:lword);

This procedure reads a single lword value from the blob specified by the b parameter (at offset rcursor in the blob) and stores this lword via the lwordVal reference parameter. This function call advances the value of rcursor by 16. This call is effectively equivalent to blob.read(b, lwordVal, 16); except that it does not return the number of bytes read in EAX (which is always 16, assuming there are no exceptions).

It will raise an ex.BlobOverflow exception if the value of rcursor is greater than or equal to the current blob length.

```
HLA high-level calling sequence example:

blob.getLword( blobPointer, lwordVar );


HLA low-level calling sequence example:
```

```
push( blobPointer );
pushd( &lwordVar );
call blob.getLword;
```

5.14 Blob Output Routines

The output routines in the blobs module are very similar to the file output routines in the fileio module as well as the output routines in the stdout library module. In general, these routines require (at least) two parameters; the first is the (pointer to the) blob object, the second parameter is usually the value to write to the blob. Some functions contain additional parameters that provide formatting information.

All output is written to the blob at the wcursor offset into the blob's data. For each byte written, wcursor is incremented by 1. If wcursor's value becomes greater than or equal to the blob's length value, then length is adjusted as well (that is, length and wcursor will have the same value). If wcursor exceeds the value in the maxlen field, then these functions raise an ex.BlobOverflow exception.

See the descriptions of the corresponding functions in the fileio module for more details.

Note that function names of the form blob.cat* are synonyms for the blob.put* functions.

```
blob.newln( b:blob.t )
blob.putbool( b:blob.t; b:boolean )
blob.putc( b:blob.t; c:char )
blob.putcSize( b:blob.t; c:char; width:int32; fill:char )
blob.putcset( b:blob.t; cst:cset   )
blob.puts( b:blob.t; s:string  )
blob.putsSize(  b:blob.t; s:string; width:int32; fill:char )
blob.putb( b:blob.t; b:byte  )
blob.puth8( b:blob.t; b:byte  )
blob.puth8Size( b:blob.t; b:byte; size:dword; fill:char )
blob.putw( b:blob.t; w:word )
blob.puth16( b:blob.t; w:word  )
blob.puth16Size( b:blob.t; w:word; size:dword; fill:char )
blob.putd( b:blob.t; d:dword )
blob.puth32( b:blob.t; d:dword )
blob.puth32Size( b:blob.t; d:dword; size:dword; fill:char )
blob.putq( b:blob.t; q:qword )
blob.puth64( b:blob.t; q:qword )
blob.puth64Size( b:blob.t; q:qword; size:dword; fill:char )
blob.puttb( b:blob.t; tb:tbyte )
blob.puth80( b:blob.t; tb:tbyte )
blob.puth80Size( b:blob.t; tb:tbyte; size:dword; fill:char )
blob.putl( b:blob.t; l:lword )
blob.puth128( b:blob.t; l:lword )
blob.puti8 ( b:blob.t; b:byte  )
blob.puti8Size ( b:blob.t; b:byte;  width:int32; fill:char )
blob.puti16( b:blob.t; w:word  )
blob.puti16Size( b:blob.t; w:word;  width:int32; fill:char )
blob.puti32( b:blob.t; d:dword )
blob.puti32Size( b:blob.t; d:dword; width:int32;  fill:char )
blob.puti64( b:blob.t; q:qword )
blob.puti64Size( b:blob.t; q:qword; width:int32; fill:char )
blob.puti128( b:blob.t; l:lword )
blob.puti128Size( b:blob.t; l:lword; width:int32; fill:char )
blob.putu8 ( b:blob.t; b:byte  )
blob.putu8Size( b:blob.t; b:byte; width:int32; fill:char )
blob.putu16( b:blob.t; w:word  )
blob.putu16Size( b:blob.t; w:word; width:int32; fill:char )
blob.putu32( b:blob.t; d:dword )
blob.putu32Size( b:blob.t; d:dword; width:int32; fill:char )
blob.putu64( b:blob.t; q:qword )
```

```
blob.putu64Size( b:blob.t; q:qword; width:int32; fill:char )
blob.putu128( b:blob.t; l:lword )
blob.putu128Size( b:blob.t; l:lword; width:int32; fill:char )
blob.pute32( b:blob.t; r:real32; width:uns32 )
blob.pute64( b:blob.t; r:real64; width:uns32 )
blob.pute80( b:blob.t; r:real80; width:uns32 )
blob.putr32( b:blob.t; r:real32; width:uns32; decpts:uns32; pad:char )
blob.putr64( b:blob.t; r:real64; width:uns32; decpts:uns32; pad:char )
blob.putr80( b:blob.t; r:real80; width:uns32; decpts:uns32; pad:char )
blob.put( list_of_items )
```

5.15 Blob Input Routines

The input routines in the blobs module are very similar to the file input routines in the fileio module as well as the input routines in the stdin library module. In general, these routines require one parameter: the pointer to the blob object.

All input is read from the blob starting at the rcursor offset into the blob's data. For each byte read, rcursor is incremented by 1. If rcursor's value becomes greater than or equal to the blob's length value, then these functions raise an ex.EndOfFile exception.

See the descriptions of the corresponding functions in the fileio module for more details.

```
blob.readLn( b:blob.t );
blob.eoln( b:blob.t ); @returns( "al" );
blob.getc( b:blob.t );  @returns( "al" );
blob.gets( b:blob.t; s:string );
blob.a_gets( b:blob.t ); @returns( "eax" );
blob.geth8( b:blob.t ); @returns( "al" );
blob.geth16( b:blob.t ); @returns( "ax" );
blob.geth32( b:blob.t ); @returns( "eax" );
blob.geth64( b:blob.t );@returns( "edx:eax" );
blob.geth80( b:blob.t; var dest:tbyte );
blob.geth128( b:blob.t; var dest:lword );
blob.geti8( b:blob.t ); @returns( "al" );
blob.geti16( b:blob.t ); @returns( "ax" );
blob.geti32( b:blob.t ); @returns( "eax" );
blob.geti64( b:blob.t );@returns( "edx:eax" );
blob.geti128( b:blob.t; var dest:lword );
blob.getu8( b:blob.t ); @returns( "al" );
blob.getu16( b:blob.t );  @returns( "ax" );
blob.getu32( b:blob.t ); @returns( "eax" );
blob.getu64( b:blob.t );@returns( "edx:eax" );
blob.getu128( b:blob.t; var dest:lword );
blob.getf( b:blob.t ); @returns( "st0" );
blob.get( List_of_items_to_read );
```