21 Memory-Mapped I/O (mmap.hhf)

The HLA Standard Library provides a set of routines for creating and manipulating memory-mapped files. Memory-mapped files are very efficient because they use the underlying operating systems' virtual memory subsystem for file I/O. When you open a memory-mapped file, the OS maps the entire file into the process' address space. Reading data from the file consists of nothing more than a memory access. Indeed, random file access is trivial in a memory mapped file system (you can treat the entire file as one huge array of characters from your software's point of view).

Although memory-mapped file access is very fast, the HLA Standard Library implementation does have a couple of limitations that make it unacceptable for some applications. First of all, as the operating system maps the file into your process' address space, memory-mapped files cannot exceed 2GBytes in size (in fact, the operating system might not even support files that large). Second, when processing existing files, you cannot extend the file's size. You may modify any data that already exists in the file, but you cannot append data to the end of the file. Third, when creating a new file, you must specify the size of the file when you first create it. You cannot open the file and then arbitrarily extend it during program execution as you can with a standard file.

21.1 MMAP Module

To call functions in the MMap module, you must include one of the following statements in your HLA application:

```
#include( "mmap.hhf" )
or
#include( "stdlib.hhf" )
```

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

21.2 Class Fields

Note that the memory-mapping module in the HLA Standard Library is implemented as a class. This class (mmap) defines the following public fields:

`filePtr:dword;`

This field holds a pointer to the first byte of the file in memory. You must not access any data in the file prior to this address. When you create a mmap_t object (but haven't yet opened a memory mapped file), or after you close the memory-mapped file (using the close method described below), the mmap_t class initializes this field with NULL.

`fileSize:dword;`

This field holds the size of the file when it is mapped into memory. The memory mapping module initializes this field with zero when you don't have a currently opened memory-mapped file.

`endFilePtr:string;`

This field holds a pointer to the first byte beyond the end of the file in memory. You must not access any data in the file equal to or beyond this address. When you create a mmap_t object (but haven't yet opened a memory mapped file), or after you close the memory-mapped file (using the close method described below), the mmap_t class initializes this field with NULL.

The mmap_t class also contains several private fields. Your applications must not modify the values of these private fields. The class does provide accessor methods if you wish to test the values of these private fields.

21.3 Class Procedures and Methods

Because the HLA stdlib implements the mmap_t functions as a class, this document will not provide low-level calling sequence examples (which aren't especially practical for object-oriented function calls). Those who insist on making low-level calls to these functions should consult the HLA reference manual for information on making direct (low-level) calls to object-oriented functions.

procedure mmap_t.create(); @returns("ESI");

This procedure is the static class constructor. If you call this procedure using the class name (i.e., mmap_t.create();) then this constructor will allocate storage for a new mmap_t object on the heap, initialize that object, and return a pointer to the object in the ESI register. If you call this procedure via an object variable reference (e.g., mmapVar.create();) then this procedure will simply initialize the fields of that object.

As with all objects in HLA, you must call the mmap_t.create constructor before using the object. Failure to do so will cause the system to crash whenever you attempt to call any of this class' methods.

```
HLA high-level calling sequence example:

mmap_t.create( );
mov( esi, mmapObjPtr );
```

method mmap_t.destroy();

This is the class destructor. It deinitializes the mmap_t object, closes any memory-mapped file left open, and deallocates the storage for the object if it was allocated on the heap. Note that you should not rely upon the destructor to close your memory-mapped files - you should always explicitly call the *mmap_t.close* method to do this.

Because mmap_t.destroy is a method, you must only call this function after initializing some mmap_t object and you must only call this function via the object invocation mechanism. If you try to call mmap_t.destroy on an uninitialized mmap_t object, or if you try to call mmap_t.destroy directly, you will likely cause a program crash.

```
HLA high-level calling sequence examples:

mmapObjPtr.destroy();
mmapStaticVar.destroy();
```

method mmap_t.openNew(filename:string; maxSize:dword);

This method opens a new memory-mapped file. The filename parameter specifies the name of the file on the disk. If the file already exists, this call will delete the file before opening a new file by that name. The filename string must be a valid pathname. The maxSize parameter specifies the size of the file (in bytes) that this call will create. You must specify the size of the memory-mapped file when you open it. This method updates the object's fields, including the filePtr, endFilePtr, and fileSize fields. This procedure does not return the pointer to the file in EAX, use the filePtr field to obtain the address of the mapped file object. Note that this method always opens the file for reading and writing.

Because *mmap_t.openNew* is a method, you must only call this function after initializing some mmap_t object and you must only call this function via the object invocation mechanism. If you try to call *mmap_t.openNew* on an uninitialized mmap_t object, or if you try to call *mmap_t.openNew* directly, you will likely cause a program crash.

```
HLA high-level calling sequence examples:

mmapObjPtr.openNew( "AMemMappedFile", 8192 );
mmapStaticVar.openNew( "AnewFile", 16384 );
```

method mmap_t.open(filename:string; Access:dword);

This method maps an existing file into the process' address space. The filename parameter is a string specifying the pathname of the file to open. The Access parameter is either fileio.r or fileio.rw and specifies whether you're opening the file as a read-only or read/write object. This call maps the entire file into the process' address space (assuming the file is small enough to fit into the address space, of course). This method call initializes the filePtr, endFilePtr, and fileSize fields of the object as appropriate for the file.

Because *mmap_t.open* is a method, you must only call this function after initializing some mmap_t object and you must only call this function via the object invocation mechanism. If you try to call *mmap_t.open* on an uninitialized mmap_t object, or if you try to call *mmap_t.open* directly, you will likely cause a program crash.

```
HLA high-level calling sequence examples:

mmapObjPtr.open( "AMemMappedFile", fileio.r );
mmapStaticVar.open( "AnExistingFile", fileio.rw );
```

method mmap_t.close();

This method unmaps the file and closes it. It also resets the object's fields to their default values (e.g., filePtr=NULL, endFilePtr=NULL, and fileSize=0). You may not access data in the memory mapped file after closing the file. Note that you may re-open the same (or a different) file using *mmap_t.open* or *mmap_t.openNew* after you close a file (and you don't need to call *mmap_t.create* unless you also call *mmap_t.destroy* after calling *mmap_t.close*).

Because *mmap_t.close* is a method, you must only call this function after initializing some mmap_t object and you must only call this function via the object invocation mechanism. If you try to call *mmap_t.close* on an uninitialized mmap_t object, or if you try to call *mmap_t.close* directly, you will likely cause a program crash.

```
HLA high-level calling sequence examples:

mmapObjPtr.close();
mmapStaticVar.close();
```

method mmap_t.getFileName();

This is an accessor function that returns the filename string (pointer) in the EAX register. The program must not modify this string in any way. Note that this is a pointer to the string data held in the object itself, this is not a copy of the string. You should make a copy of this string if you intend to modify its data.

Because *mmap_t.getFileName* is a method, you must only call this function after initializing some mmap_t object and you must only call this function via the object invocation mechanism. If you try to call *mmap_t.getFileName* on an uninitialized mmap_t object, or if you try to call *mmap_t.getFileName* directly, you will likely cause a program crash.

```
HLA high-level calling sequence example:

mmapObjPtr.getFileName();
mov( eax, fileNameString );
```

method mmap_t.getOpen();

This is an accessor function that returns a boolean value in AL: false if the file is not currently open, true if there is a file mapped into the process' address space. This field is only valid after you call *mmap_t.create*.

Because *mmap_t.getOpen* is a method, you must only call this function after initializing some mmap_t object and you must only call this function via the object invocation mechanism. If you try to call *mmap_t.getOpen* on an uninitialized mmap_t object, or if you try to call *mmap_t.getOpen* directly, you will likely cause a program crash.

```
HLA high-level calling sequence example:

mmapObjPtr.getOpen();
if( al ) then

  // Do something if the file is open

  endif;
```

method mmap_t.getMalloc();

This is an accessor function that returns true if the object is allocated dynamically on the heap, it returns false if the object is a static or automatic variable.

Because *mmap_t.getMalloc* is a method, you must only call this function after initializing some mmap_t object and you must only call this function via the object invocation mechanism. If you try to call *mmap_t.getMalloc* on an uninitialized mmap_t object, or if you try to call *mmap_t.getMalloc* directly, you will likely cause a program crash.

```
HLA high-level calling sequence example:

mmapObjPtr.getOpen();
if( al ) then

  // Do something if the file is open

  endif;
```