36 Threads Module (threads.hhf)

The HLA Threads module provides a set of routines that let you create, control, and synchronize multiple threads in an application.

A Note About Thread Safety: While the routines in the thread library are (mostly) thread safe, keep in mind that you must be linking in a thread-safe version of the HLA Standard Library if you expect calls to other functions to operate in a thread-safe manner.

36.1 Threads Module

To use the thread functions in your application, you will need to include the following statement at the beginning of your HLA application:

#include("threads.hhf")

Note that the "stdlib.hhf" header file does not automatically include the threads.hhf header file. This is because simply including the "threads.hhf" header file may force the inclusion of considerable code, even if you do not call any functions in the thread library. Therefore, you must explicitly include the threads.hhf header file if you want to call thread functions in the HLA Standard Library.

If you are using thread functions in your application, you must use the "-thread" command-line parameter to force HLA to link in the thread-safe version of the HLA Standard Library. Failure to do so will probably cause the linker to fail; even if you manage to get the program to link properly, you'll link in non-thread-safe versions of the HLA Standard Library functions and this will probably cause your program to fail or otherwise misbehave.

The functions in the threads module are broken down into six types: thread creation, thread identification, thread local storage, events, critical section maintenance, and semaphore maintenance. The following sections will describe each of these categories.

36.2 Thread Creation

```
procedure thread.create( func:threadFunc_t; parm:dword; stackSize:dword );
    @returns( "eax" );
```

This function creates a new thread. The *func* parameter is the address of an HLA procedure where the thread will begin execution. This function has the following prototype:

```
type
  threadFunc_t:procedure( parm:dword );
```

That is, the thread function must have a single double word parameter.

The *parm* argument (to *thread.create*) is passed along to the thread function specified by the *func* argument. This can be any 32-bit value you want. Often, this argument is a pointer to some global data you're supplying to the thread. Keep in mind, however, that the thread may not begin executing before *thread.create* returns to its caller. You must ensure that any data whose address you pass in the *parm* argument remains valid as long as the new thread requires that data. In particular, do no pass the address of some local variables allocated on the stack that might go away when the procedure that calls *thread.create* returns to its caller.

The *stackSize* parameter specifies the number of bytes of storage that will be allocated for the stack when the thread is created by the operating system. This value should be a multiple of 4,096 bytes. If you specify zero, the system will assign a default value (the default value is OS dependent). Unless your thread requires very little thread storage, you should always supply a value for the *stackSize* parameter. Note that the HLA Standard Library will allocate storage for its own thread local variables on the stack created for the thread. Therefore, you should allocate an additional 4,096 bytes above and beyond your own needs to provide sufficient storage for the stdlib thread local objects.

The thread creation function returns a thread identifier in the EAX register. This information is useful when you have multiple threads executing the same code (that is, the same thread function) and you need to pass information to a specific thread. The individual threads can determine their own thread ID and use that to determine whether a packet of information is intended for them (see the discussion of the *thread.getCurrentThreadHandle* function for details).

A thread terminates execution by returning from the thread function. If you want to prematurely terminate a thread, then that thread must jump to the end of the procedure and return from it (or clean up the stack and execute a RET instruction). Note that the HLA Standard Library does not allow one thread of execution to terminate another thread (including the parent thread that started the thread in the first place). If you want to terminate one thread under the control of another thread, then you must pass some message to that thread and tell it to terminate itself.

Each thread of execution has its own exception handling system. Exceptions that the system or program raises in one thread must be handled by that thread. There is no way to pass exceptions on to a different thread (including the parent thread). If an exception occurs in a thread and you do not provide an exception handler (via a **try..endtry** statement), then the system will abort execution of the whole application.

The following examples assume the presence of the following thread function:

```
procedure myThreadFunc( theParm:dword );
var
  myThreadID:dword;
begin myThreadFunc;

  try

      thread.getCurrentThreadHandle();
      mov( eax, myThreadID );

      // Code that does something for this thread

  anyexception

      stdout.put
      (
          "Thread $",
          myThreadID,
          " terminated with exception $",
          eax,
          nl
      );

  endtry;

end myThreadFunc;


HLA high-level calling sequence examples:

  thread.create( &myThreadFunc, 0, 0 );  // Default stack size
  mov( eax, childThreadID );


HLA low-level calling sequence examples:

  pushd( &myThreadFunc );
  pushd( 0 );                       // parm value = 0
  pushd( 0 );                       // Default stack size
  call thread.create;
  mov( eax, childThreadID );
```

36.3 Thread Identification

```
procedure thread.getCurrentThreadHandle;  @returns( "eax" );
```

This function returns a unique thread identifier value in the EAX register for the current thread. This value matches the value that *thread.create* returns (for the child thread, assuming that the child thread is the one calling *thread.getCurrentThreadHandle*). You should not assume that the value that this function returns is the same as the thread ID used by the underlying OS.

```
HLA high-level calling sequence examples:

  thread.getCurrentThreadHandle();
  mov( eax, myThreadID );

HLA low-level calling sequence examples:

  call thread.getCurrentThreadHandle;
  mov( eax, myThreadID );
```

36.4 Thread Local Storage

Sometimes various procedures in a thread need to communicate data amongst themselves using static storage. Unfortunately, you must be very careful about using static objects in a threaded application. In general, using static objects renders the application thread unsafe (that is, causes the program to fail when two or more threads attempt to use the same static object).

A good example of this problem occurs in the HLA Standard Library. Consider the *underscores* variable that determines if a hexadecimal numeric conversion routine should emit underscores between groups of four hexadecimal digits. This is a global value that all of the hexadecimal conversion routines in the Standard Library reference. Now suppose you have two threads that call these conversion routines. If one thread turns underscore output on and the other thread turns it off in the middle of a conversion in the other thread, the conversion will be incorrect. The solution is to give each thread its own copy of the underscores variable. When one thread turns this feature one, it does not affect the conversions in any of the other threads. Unfortunately, giving each thread its own copy of the *underscores* variable is a lot more difficult that it sounds. You cannot use static objects for this purpose – the different threads will all access the same static objects. You cannot allocate the *underscores* variable on the stack in your thread, the various procedures won't be able to access that object without knowing its exact address in memory.

The solution is *thread local storage*. With thread-local storage (TLS) you request (once) a TLS context handle from the operating system. With this handle you can store a 32-bit thread-local value and retrieve that 32-bit thread-local value. Normally, you don't store the actual thread-local data via this handle, instead you store a pointer to a data structure containing all the thread-local data. This structure can be allocated on the heap or you can allocate it in the VAR section of your main thread. Because you can retrieve the address of this data structure via TLS calls, various functions can figure out the address of the global objects they're interested in via some offset from the address of the structure.

Note that you only need to obtain a single TLS context handle – you do not obtain a separate handle for each thread in your application. Your main thread should obtain this context handle before spawning any other child threads. You should store the value of this context handle in a global, static, object that all threads can access. You will never modify this object directly; instead, you will pass the address of the object to the *thread.createTLS* function and then your threads will only read this value from that point forward (which is a safe use of a global, static, object in a threaded application).

```
procedure thread.createTLS( var context:dword );
```

This function asks the operating system to create a thread-local storage context handle. You pass the address of the context variable to this function as the single parameter and the *thread.createTLS* function will automatically (and atomically) update that variable. Note that you should create (initialize) the context handle before creating any threads that might use it. Otherwise you might create a race condition where the main thread stops, a child thread runs, and the child thread attempts to use the context handle before the main thread initializes it.

```
HLA high-level calling sequence examples:

  thread.createTLS( contextHandle );
```

```
HLA low-level calling sequence examples:

  lea( eax, contextHandle );
  push( eax );
  call thread.createTLS;
```

procedure thread.setTLS(context:dword; valueToSet:dword);

The *thread.setTLS* function stores the value found in the *valueToSet* parameter into a thread-local double word value specified by the *context* parameter. Generally, you will pass the address of some block of memory (some data structure) as the *valueToSet* parameter. If you allocate that block of memory on the heap (e.g., via *mem.alloc*) or on your local stack, any function in the current thread can access that data structure later by calling the *thread.getTLS* function.

Although you can use *thread.setTLS* to set a single variable value (rather than setting the address of some data structure allocated for the current thread), you can only access 32 bits of data this way. As a result, most programmers store an address to some data structure via *thread.setTLS* rather than directly storing data.

If you use *thread.setTLS* in the normal manner, by allocating some storage and storing an address away, you should allocate the storage at the very beginning of your thread's main function. If this is a fixed-size data structure, you can allocate it as part of your local variables (in the VAR section) and simply take the address of the structure (e.g., with the LEA instruction) and pass that as the *valueToSet* argument. If the data structure is variable in size, then *mem.alloc* is probably a good choice for allocating the data structure.

```
HLA high-level calling sequence examples:

  lea( eax, myLocalData );
  thread.setTLS( contextHandle, eax );

HLA low-level calling sequence examples:

  push( contextHandle );
  lea( eax, myLocalData );
  push( eax );
  call thread.setTLS;
```

procedure thread.getTLS(context:dword); @returns("eax");

The *thread.getTLS* function returns the value associated with the thread context handle passed as the single argument. This value is set via an earlier call to the *thread.setTLS* function. See the discussion of *thread.setTLS* for more details.

```
HLA high-level calling sequence examples (assuming the thread local storage value
has been set to point at an object of type "someDataType" by a previous call to
thread.setTLS):

  thread.getTLS( contextHandle );
  mov( (type someDataType [eax]).someField, ecx );// Retrieve data

HLA low-level calling sequence examples:

  push( contextHandle );
  call thread.getTLS;
  mov( (type someDataType [eax]).someField, ecx );// Retrieve data
```

36.5 Events

The HLA Standard Library threads module provides a thread synchronization system known as events. An event is something that a thread can wait upon until a different thread signals (or sets) the event. An HLA stdlib event is an object that various threads can use to coordinate events in the program.

To use events, a program must first create an event object via a call to *thread.createEvent*. This initializes the event and puts it in the non-signaled/non-set state. The *thread.createEvent* returns an OS event handle that you will save to allow various threads to work with that event object. When you are done using a thread, you can tell the OS to reclaim the resources used by the event object.

When you create an event, it is initialized in an unsignaled state. Whenever some thread waits on an event (by calling *thread.waitForEvent*), that thread will block (suspend) until some other thread signals the event by calling *thread.setEvent*. If two or more threads are waiting for an event, only one thread will be placed in the executable state when an event is signaled. Additional calls to *thread.setEvent* will be necessary to resume any other threads waiting for the event.

Whenever a thread waiting on an event resumes execution after that event has been signaled, the system automatically sets the event to the unsignaled state. There is no explicit call you can make to "unsignal" or unset an event.

```
procedure thread.createEvent; @returns( "eax" );
```

The *thread.createEvent* function creates an event object and returns a handle to that object in the EAX register. All events you use must be initialized via a call to *thread.createEvent*. Note that event initialization consumes resources internal to the OS' thread library. You should call *thread.deleteEvent* to reclaim those resources when you are done using the event object you've created via *thread.createEvent*.

```
HLA high-level calling sequence examples:

  thread.createEvent();
  mov( eax, eventHandle );

HLA low-level calling sequence examples:

  call thread.createEvent;
  mov( eax, eventHandle );
```

```
procedure thread.deleteEvent( event:dword );
```

The *thread.deleteEvent* function reclaims all OS/library resources in use by an event object. You should call this function after you are done using an event object. You must not call this function on an event handle if any threads are waiting on the event. Of course, you must not continue to use the event handle after deleting the event.

```
HLA high-level calling sequence examples:

  thread.deleteEvent( eventHandle );

HLA low-level calling sequence examples:

  push( eventHandle );
  call thread.deleteEvent;
```

```
procedure thread.setEvent( event:dword );
```

The *thread.setEvent* function signals the occurrence of an event. If some thread is waiting for this event to occur, this call will resume the execution of that thread. If more than one thread is waiting on the event, then only one thread will be unblocked and allowed to execute. In order to release all threads waiting on an event, you must call *thread.setEvent* once for each blocked thread. It is the application's responsibility to count the number

of threads waiting on an event and call *thread.setEvent* an appropriate number of times. Calling *thread.setEvent* multiple times without having any threads waiting on the event between signaling the event is undefined. Some OS thread APIs might count the number of calls and release that many threads that (ultimately) wait for the event. Other OS thread packages might ignore multiple requests and release only one thread that waits on the event. Still other OSes may completely ignore the call to *thread.setEvent* if there are no threads waiting on that particular event.

```
HLA high-level calling sequence example:

  thread.setEvent( eventHandle );

HLA low-level calling sequence examples:

  push( eventHandle );
  call thread.setEvent;
```

procedure thread.waitForEvent(event:dword);

The *thread.waitForEvent* function blocks (suspends the execution of) the current thread until some other thread signals (sets) the event with a call to *thread.setEvent*. Note that the results are undefined if a call is made to *thread.setEvent* prior to some other thread waiting on that event. The system may choose to ignore the earlier call to *thread.setEvent* or it may immediately resume the thread and return from a call to *thread.setEvent*. The exact semantics are OS-dependent.

```
HLA high-level calling sequence example:

  thread.waitForEvent( contextHandle );

HLA low-level calling sequence examples:

  push( contextHandle );
  call thread.waitForEvent;
```

36.6 Critical Sections

Critical sections are synchronization objects that ensure that only one thread at a time executes a protected section of code (or accesses some data structure). A thread *enters* and *leaves* a critical section. While one thread is holding a critical section lock, an attempt by some other thread to enter that same critical section causes the second thread to block until the first thread leaves the critical section.

As for all synchronization objects the HLA stdlib supports, an application must first create a critical section object to obtain a handle to be used when entering and leaving critical sections. Because the creation of a critical section allocates some system resources, the application should delete the critical section object when it is done using it. The *thread.createCriticalSection* and *thread.deleteCriticalSection* functions handle these chores.

Once you've created a critical section object via *thread.createCriticalSection*, you can synchronize threads using the *thread.enterCriticalSection* and *thread.leaveCriticalSection* functions. To protect a sequence of instructions (that, perhaps, operate on a protected data structure) you would call the *thread.enterCriticalSection* (passing it the handle of a critical section object you've created) to lock the use of that particular critical section object. When you are done executing the protected code, you call the *thread.leaveCriticalSection* function to release the lock. If any other thread attempts to enter the same critical section (by calling *thread.enterCriticalSection* and passing in the same critical section handle), then that second thread will block until the first thread calls *thread.leaveCriticalSection* and releases the lock.

If one thread is holding a critical section lock and two or more additional threads attempt to enter the same critical section, all those new threads will block. When the thread holding the lock calls *thread.leaveCriticalSection*, only one of the waiting threads will be activated to resume execution. Note that the

order of thread activation is not defined and you should not assume that the first blocked thread will be the first to be released. The only guarantee is that exactly one thread will be released.

procedure thread.createCriticalSection; @returns("eax");

The *thread.createCriticalSection* function creates a critical section object and returns a handle to that object in the EAX register. All critical sections you use must be initialized via a call to *thread.createCriticalSection*. Note that critical section initialization consumes resources internal to the OS' thread library. You should call *thread.deleteCriticalSection* to reclaim those resources when you are done using the critical section object you've created via *thread.createCriticalSection*.

```
HLA high-level calling sequence examples:

  thread.createCriticalSection();
  mov( eax, csHandle );

HLA low-level calling sequence examples:

  call thread.createCriticalSection;
  mov( eax, eventHandle );
```

procedure thread.deleteCriticalSection(csHandle:dword);

The *thread.deleteCriticalSection* function reclaims all OS/library resources in use by a critical sectionobject. You should call this function after you are done using a critical section object. You must not call this function on a critical section handle if any threads are executing in the critical section. Of course, you must not continue to use the critical section handle after deleting the critical section.

```
HLA high-level calling sequence examples:

  thread.deleteCriticalSection( csHandle );

HLA low-level calling sequence examples:

  push( csHandle );
  call thread.deleteCriticalSection;
```

procedure thread.enterCriticalSection(csHandle:dword);

The *thread.enterCriticalSection* function first checks to see if some other thread has already entered the critical section specified by the *csHandle* parameter. If this is the case, then the current thread (that is calling *thread.enterCriticalSection*) blocks until the first thread releases the critical section handle (that is, it leaves the critical section) by calling *thread.leaveCriticalSection*. If no other thread currently holds the critical section lock, or if the current thread resumes execution because some other thread releases the lock (by calling *thread.leaveCriticalSection*), then the current thread obtains the lock and execution resumes with the first instruction after the call to *thread.enterCriticalSection*.

```
HLA high-level calling sequence examples:

  thread.enterCriticalSection( csHandle );

HLA low-level calling sequence examples:

  push( csHandle );
  call thread.enterCriticalSection;
```

```
procedure thread.leaveCriticalSection( csHandle:dword );
```

The *thread.leaveCriticalSection* function releases the critical section lock specified by the *csHandle* parameter. This allows any other thread that is waiting on the critical section to resume execution (and obtain the critical section lock).

```
HLA high-level calling sequence examples:

  thread.leaveCriticalSection( csHandle );

HLA low-level calling sequence examples:

  push( csHandle );
  call thread.leaveCriticalSection;
```

36.7 Semaphores

Semaphores are the generic process/thread synchronization mechanism. Semaphores provide two main extensions over other sychronization objects provide by the HLA stdlib:

- HLA stdlib semaphores allow synchronization of processes (e.g., different applications) as well as threads.
- HLA stdlib semaphores are counting semaphores, allowing
- *n* processes access to some protected resource (where *n* is some value you specify when creating the semaphore).

Like the other HLA stdlib synchronization objects, you must create a semaphore object before using it and you must delete a semaphore object when you are done using it. The *thread.createSemaphore* and *thread.deleteSemaphore* functions handle these chores. When creating the semaphore, you specify the number of resources the semaphore will protect (that is, the number of threads that can concurrently hold the semaphore and run without blocking). You also specify a (system-wide) semaphore name when creating the semaphore; the operating system uses this name to connect semaphore objects in different processes to the same system-wide semaphore object (that is, if two processes specify the same name for the semaphore object, then those two processes will access the exact same semaphore).

To obtain a resource held be a semaphore, you will call the *thread.waitSemaphore* function. This function blocks if there are no resources available, it will decrement the resource count and return if resources are available. To release a semaphore resource that a thread is holding, the thread executes the *thread.releaseSemaphore* function.

```
procedure thread.createSemaphore( maxCnt:dword; semName:string );
    @returns( "eax" );
```

The *thread.createSemaphore* function creates a semaphore object and returns a handle to that object in the EAX register. All future access to that semaphore will be via the handle value that *thread.createSemaphore* returns in the EAX register.

The *maxCnt* parameter specifies the number of threads (or processes) that may concurrently hold the semaphore before the operating system blocks any further semaphore requests. If *maxCnt* is one, then the semaphore behaves in a manner similar to a critical section insofar as it only allows access to the lock by one thread (or process) at a time. This is known as a binary semaphore.

The *semName* string parameter provides a system-wide name for the semaphore. This string should correspond to an existing filename in the system for best results (create an empty file that has the semaphore's name if you want to create a semaphore using a name other than that of some existing file). If two different calls to the *thread.createCriticalSection* function specify the same semaphore name, then the handle value that this function returns will refer to the same semaphore object.

If multiple calls to thread.createCriticalSection specify the same string for *semName* but specify different values for *maxCnt*, then the result is undefined. The system may use the last value specified, the first value specified, or any other value it pleases.

All semaphore objects you use must be initialized via a call to *thread.createSemaphore*. Note that semaphore initialization consumes resources internal to the OS' thread library. You must call *thread.deleteSemaphore* to reclaim those resources when you are done using the critical section object you've created via *thread.createSemaphore*.

Warning: because of a known issue in the UNIX SYSV semaphore interface (that the HLA stdlib used under Linux, Mac OSX, and FreeBSD), there is a brief time period during semaphore creation between the creation of a semaphore object and the setting of the resource count where the system could be interrupted. If another process executes between these two points and specifies a different semaphore count, the results could be unexpected. This is yet another reason why you want to specify the same count value when requesting multiple semaphore handles using the same semaphore name. For this same reason, you should try to allocate all semaphore handles in your main program, before spawning any additional threads (though this won't help much if multiple processes or applications are creating the same semaphore).

```
HLA high-level calling sequence examples:

  thread.createCriticalSection();
  mov( eax, csHandle );

HLA low-level calling sequence examples:

  call thread.createCriticalSection;
  mov( eax, eventHandle );
```

procedure thread.deleteSemaphore(semHandle:dword);

The *thread.deleteSemaphore* function will (possibly) delete system resources in use by a semaphore. The exact operation of this command is system dependent. Under Windows, it will decrement a reference counter and if the current process is the last process using the semaphore, this call will free up all system resources used by the semaphore.

As this document was being written, this function is a no-operation under *NIX operating systems. For those operating systems you will need to manually delete the semaphore object using the ipcrm command (use ipcs to list the semaphores currently in use by the system). Even if you are using a *NIX operating system, you should call *thread.deleteSemaphore* because the semantics of this function might change in future versions of the HLA stdlib.

```
HLA high-level calling sequence examples:

  thread.deleteSemaphore( semHandle:dword );

HLA low-level calling sequence examples:

  push( semHandle );
  call thread.deleteSemaphore;
```

procedure thread.waitSemaphore(semHandle:dword);

The *thread.waitSemaphore* function decrements the semaphore resource count (initialized to the value of the *maxCnt* parameter by the *thread.createSemaphore* function). If the result is less than or equal to zero, then this function returns and execution continues. If the result it negative, then this function blocks the current thread until some process releases the semaphore. When the process is done using the resource protected by the semaphore, it should release that resource via a call to the *thread.releaseSemaphore* function.

If a thread wishes to grab multiple resources protected by a semaphore, it can make multiple calls to *thread.waitSemaphore*. It must make the corresponding number of calls to *thread.releaseSemaphore* to release it of those locks it allocates. Obviously, a thread should not call *thread.waitSemaphore* more than *maxCnt* times (*maxCnt* being the parameter value passed to *thread.createSemaphore*) otherwise deadlock will occur.

```
HLA high-level calling sequence examples:

  thread.waitSemaphore( semHandle:dword );

HLA low-level calling sequence examples:
```

```
    push( semHandle );
    call thread.waitSemaphore;
```

procedure thread.releaseSemaphore(semHandle:dword);

The *thread.releaseSemaphore* function increments the internal resource count associated with the semaphore specified by *semHandle*. If there are any threads or processes blocked and waiting on that semaphore, then exactly one of those threads will be placed in an active (runnable) state and allowed to continue execution. Each process/thread should have a corresponding *thread.releaseSemaphore* call for each *thread.waitSemaphore* call it makes.

```
HLA high-level calling sequence examples:

  thread.releaseSemaphore( semHandle:dword );

HLA low-level calling sequence examples:

  push( semHandle );
  call thread.releaseSemaphore;
```