# Helium 2.0 API Reference

Neil Klingensmith Control Engineering Solutions 2008-09-01

Revision Log

| Date | Action | Initials |
|---|---|---|
| 2008-09-01 | Created | NAK |
| | | |
| | | |

## Table of Contents

## Thread Management

Thread management functions are used to create and destroy threads in Helium. For more information on threads and thread scheduling, refer to the document titled *Helium 2 Scheduler*.

### threadCreate

```
THREAD *threadCreate( short  priority,
                      void (*threadPtr)(),
                      int   *stack,
                      int    stackSize,
                      void  *data )
```

Creates a thread consisting of a block of code and a stack space. The block of code is equivalent to a `main` function in a normal C program (although it should be given a different name). The function used to create the thread will execute and return to the operating system when finished, just like a standard main function. A stack space should be allocated for the thread using an array.

**Table 1: Parameter description for threadCreate.**

| | |
|---|---|
| `short priority` | Priority the thread should be run at. Threads with higher priorities will be given the option to run before threads with lower priorities. The highest priority a thread can have is 0, and the lowest is 65535. |
| `void (*threadPtr)()` | Pointer to the code section for the thread. |
| `int *stack` | Pointer to the lowest address in the stack section. This is the base address of the memory block used for the stack. |
| `int stackSize` | Length (in bytes) of the stack. The initial stack pointer will be calculated by adding the base address of the stack section to the stack size. |
| `void *data` | Optional 16-bit integer that can be passed to the new thread. This may be a mutex or FIFO handle or an application-specific data structure. |

`threadCreate` is an O(n) operation, where n is the number of threads in the ready list.

### threadDestroy

```
void threadDestroy( THREAD *t )
```

Destroys a thread. The thread is removed from the list(s) it is linked into, and any pool memory that is still allocated to it is freed. Finally, the scheduler is invoked, and the next thread is executed. A thread may implicitly call this function to destroy itself by returning to the operating system.

`threadDestroy` is an O(1) operation.

### sleep

```
void sleep( unsigned short numTicks )
```

Causes the current thread to wait for a specified number of clock ticks. While the thread is waiting, other threads will be given control of the CPU.

## Dynamic Memory Allocation

### heapAlloc

```
void *heapAlloc(int size)
```

Allocates a block of memory from the memory pool of length `size` and returns a pointer to the lowest address in the memory block. In the standard Helium distribution, blocks of 8, 16, 32, and 64 bytes may be allocated. Requests for blocks of memory that do not have exactly these sizes will cause the memory allocator to return a pointer to a block of the next largest size. For example, if a block of 25 bytes is requested, the memory allocator will return a pointer to a 32-byte block. If a memory block greater than 64 bytes is requested, the memory allocator will return -1.

`heapAlloc` is an O(1) operation.

### heapFree

```
void heapFree(void *b)
```

Frees a block of memory that was allocated with `heapAlloc`.

`heapFree` is an O(n) operation where n is the total number of memory blocks available to the system.

## Mutexes

### mutexCreate

```
MUTEX *mutexCreate()
```

Creates a mutex and returns a handle to the mutex. All subsequent accesses to the mutex will be referenced using the mutex handle returned by `mutexCreate`.

`mutexCreate` is an O(1) operation.

### mutexTake

```
void mutexTake(MUTEX *m)
```

Request access to mutex `m`. If another thread is already using that mutex, the requesting thread will block until the other thread has finished using it. If the thread that is currently using the mutex has a lower priority than the requesting thread, the first thread will be promoted to the priority of the requesting thread until it has finished using the mutex.

`mutexTake` is an O(1) operation.

### mutexRelease

```
int mutexRelease(MUTEX *m)
```

Release the mutex `m`. The next thread in the mutex waiting list will be added to the ready list. The current thread will be allowed to continue executing until the next clock tick, at which time the scheduler will choose which thread to run next.

`mutexRelease` is an O(1) operation.

## FIFO Buffers

### fifoCreate

```
FIFO *fifoCreate(int mode, int numElements)
```

Creates a FIFO buffer and returns a handle to that buffer. Allows the calling thread to access that buffer as either the reading thread or the writing thread by requesting the appropriate mode. Also, the calling thread may place a limit on the number of data elements the FIFO may hold at any given time. This is used to stop the FIFO from using all available pool memory if the rate of data being written to the FIFO exceeds the rate of data being read from the FIFO. This feature can be overridden by passing NULL as the second parameter.

`fifoCreate` is an O(1) operation.

**Table 2: Parameter description for fifoCreate.**

| | |
|---|---|
| `int mode` | Indicates that the thread that creates the FIFO will either read from or write to the FIFO. Allowable modes are `FIFO_READ_MODE` or `FIFO_WRITE_MODE`. |
| `int numElements` | Maximum number of data elements the FIFO may contain. Passing NULL for this parameter puts no limit on the number of elements, making it possible for the FIFO to use as much pool memory as it needs. |

### fifoRegister

```
int fifoRegister( FIFO *f, THREAD *t, int mode )
```

Registers a thread as either a reader or a writer to the specified FIFO. As with `fifoCreate`, the mode may be either `FIFO_READ_MODE` or `FIFO_WRITE_MODE`. **Both fifoCreate and fifoRegister should be called before attempting to access data in a FIFO.**

`fifoRegister` is an O(1) operation.

## fifoWrite

```
int fifoWrite(FIFO *f, int data)
```

Writes a two-byte value to the specified FIFO buffer. The thread that calls `fifoWrite` must be registered as the writing thread for the FIFO for this call to succeed. A thread may register itself for write access using either the `fifoRegister` or `fifoCreate` system calls (see above). The return value indicates success or failure mode as outlined in Table 1 below.

**Table 3: Return values from fifoWrite.**

| Return Value from `fifoWrite` | Meaning |
|---|---|
| 0 | Operation successful. |
| -1 | `currThread` is not registered as the FIFO's writer. |
| -2 | No memory is available to allocate a new FIFO data block. |

## fifoRead

```
int fifoRead( FIFO *f, int *data )
```

Reads a two-byte value from the specified FIFO buffer. The thread that calls `fifoRead` must be registered as the reading thread for the FIFO for this call to succeed. A thread may register itself for read access using `fifoRegister` or `fifoCreate` system calls (see above).