

JN External API

UCSC-CRL-97-17

Bruce R. Montague[†]
Computer Science Department
University of California, Santa Cruz
brucem@cse.ucsc.edu

6 August 1997

Abstract

The Application Program Interfaces (APIs) developed for the UCSC Java Nanokernel (JN) are described. These APIs provide an interface to a small embedded operating system developed at UCSC which runs on a ‘single-chip’ PC and supports a web server and web camera. These APIs include the APIs specified in the public JavaSoft JDK host porting interface (*sys_api.h*). These APIs enable execution of version 1.0.1 of the Java Virtual Machine (JVM). The implementation of these APIs in 1996 was independent of the Java Green Threads API implementation. The function, arguments, and return values of each API are described.

keywords: Java, JavaOS, JN, API.

1 Introduction

This document describes the Java Nanokernel (JN) Application Program Interface (API). The Java Nanokernel is a small kernel implemented primarily to provide stand-alone support for version 1.0.1 of the Java Virtual Machine (JVM). The JVM is a multithreaded interpreter which executes the Java programming language. As originally implemented, the JVM depended on a multithreading C-runtime called *Green Threads*. The functions described in this document were developed by linking Java without the Green Threads library, and then guessing at the missing routine’s functionality by inspecting the calls made to the missing routines in the JVM source. Since much of the functionality is very conventional, this reverse engineering process produced a system that almost worked. The initial implementation was done deliberately without inspection of the Green Thread source so as not to be unduly biased by its implementation approach; when bringing up the JVM on the resulting system a few Green Threads routines were later examined to resolve specific compatibility issues. Naturally, the implementation of the JN API routines differs from that of Green Threads due to the different target environments – JN runs stand-alone on bare hardware, while Green Threads assumes it is running on top of Unix or another Unix-level virtual-memory operating system.

This API provides C ‘primitives’ that implement the concurrent programming mechanisms, low-level exception handling mechanisms, and file handling mechanisms required by the JVM.

The routines documented here are intended to be called by C programs, specifically, the JVM Java Interpreter itself. However, these routines have no specific dependency on Java or the JVM, but are simply a particular interface to a small embedded kernel.

[†]Supported in part by a gift from National Semiconductor.

The API functions are divided as follows:

- Thread support.
- Monitor support.
- File support.
- Exceptions (software interrupts, that is, *signals*).
- TCP/IP socket support.
- Miscellaneous ‘Standard’ Unix Routines.
- Time support.
- Debugging support.

These routines are listed in the following table. The socket functions are ‘completely’ Unix compatible and are not described in detail in this document. Routines in the following table that are marked with an ‘*’ are not necessary in JN but are provided for compatibility with JVM on Unix, that is, in JN they exist but have no content.

Note well that a small number of routines that are realistically part of the required host porting interface (the routines needed by the JVM) are not included in *sys_api.h* and references to such routines have been removed from this document. However, most of these routines have to do with managing Unix processes, Unix asynchronous I/O, and Unix signals. Many of these routines are simply not appropriate in an embedded system and have simply been stubbed in the JN implementation.

sysThreadBootstrap	*- Turn Unix process into first thread.
sysThreadInitializeSystemThreads	*- Start clock, idle, garbage thread.
sysThreadSingle	- Run exclusive (disable concurrency (the scheduler)).
sysThreadMulti	- Enable concurrency.
sysThreadCreate	- Create a new thread.
sysThreadInit	*- Must call at start of new thread.
sysThreadExit	- Terminate thread.
sysThreadSelf	- Obtain caller's thread ID.
sysThreadYield	- Non-preemptive CPU yield.
sysThreadSuspend	- Suspend a given thread.
sysThreadResume	- Resume a suspended thread.
sysThreadSetPriority	- Set thread's priority.
sysThreadGetPriority	- Get thread's priority.
sysThreadGetBackPtr	- Get thread context's 'cookie'.
sysThreadSetBackPtr	- Set thread context's 'cookie'.
	The cookie in a JN thread points to a high-level interpreted thread context inside the JVM.
sysThreadCheckStack	- Return 1 if stack has space.
sysThreadPostException	- Trigger an exception in a thread.
sysThreadStackBase	- Return thread's stack base.
sysThreadStackPointer	- Get thread's current stack pointer.
sysThreadEnumerateOver	- Iterate a function over all threads.
sysThreadDumpInfo	- Thread dump stub.

sysMonitorInit	- Initialize a semaphore.
sysMonitorEnter	- P(). Enter a critical section.
sysMonitorExit	- V(). Leave a critical section.
sysMonitorDestroy	- V(). Leave a semaphore and determine if its still being used.
sysMonitorEntered	- True if caller owns the semaphore.
sysMonitorWait	- Internal wait for a notify or a given time.
sysMonitorNotify	- Unblock head of internal wait queue.
sysMonitorNotifyAll	- Unblock all internal waiters.
sysMonitorSizeof	- Obtain sizeof(semaphore).
sysMonitorDumpInfo	- Dump semaphore and waiters.

sysInitFD	- Initialize file descriptor.
sysMode_Line	- Read and write lines.
sysMode_Byte	- Read and write single 'raw' bytes.
sysOpen	- Open or Create a file.
sysOpenFD	- "
sysClose	- Close a file.
sysCloseFD	- "
sysRead	- Read a file.
sysReadFD	- "
sysWrite	- Write a file.
sysWriteFD	- "
sysLseek	- Set current file position.
sysLseekFD	- "
sysAvailable	- Determine bytes till end of file.
sysAvailableFD	- "

sys_set_exception	- Specify a software interrupt handler.
sysInterruptsPending	- Returns True if pending software interrupts exist for a thread.
intrLock	- Disable software interrupts within the thread.
intrUnlock	- Enable software interrupts within the thread.

htonl	- Host to network endian conversion of 32-bit long.
ntohl	- Network to host endian conversion of 32-bit long.
accept	- Server-side passive socket connect, i.e., open.
bind	- Bind a port to a socket.
close_s	- Close a socket.
connect	- Client-side active socket connect, i.e., open.
getpeername	- Get information about other side of connection.
getsockname	- Get socket information.
listen	- Specify supported number of simultaneous passive connects.
sock_shutdown	- A number of ways to close a socket, including abort.
socket	- Create a socket.
sockkick	- Force a retransmission over a connection.
socklen	- Get send or receive queue length associated with a socket.
socketpair	- Create 2 connected sockets on the same machine.
recv	- Receive data over a connected socket (usually TCP).
recvfrom	- Receive a datagram (usually UDP).
send	- Send data over a connected socket (usually TCP).
sendto	- Send a datagram (usually UDP).

jn_malloc	- Allocate memory.
sysMalloc	- "
jn_free	- Free allocated memory.
sysFree	- "
jn_realloc	- Change size of allocated memory.
sysRealloc	- "
sysCalloc	- Allocate and zero memory.
perror	- Print an error and halt.
printf	- Formated output.
fprintf	- Formated output to a file.
fstat	- Get file statistics (length).
sysStat	- "
sysMkdir	- Create a directory.
sysUnlink	- Delete a file.
strdup	- Duplicate a string.
sysPutenv	- Create an environmental variable.
sysGetenv	- Get the contents of an environmental variable.

gettimeofday	- Get the time, in seconds and milliseconds.
jn_gmt_tm_to_time_t	- Convert 'calendar time' to Unix 'epoch' time.
jn_gmt_dmyhms_to_epoch_UTC	- " without the need for an intermediate structure.
jn_julian_day_to_gregorian_dmy	- Convert Julian day to 'calendar time'.
jn_gregorian_dmy_to_julian_day	- Convert a 'calendar day' to a Julian day.

sysCollectChecksum	- Checksum all the code in the system.
--------------------	--

2 Thread Calls

This section describes Thread APIs. These APIs are rather conventional light-weight multithreading primitives.

```
int sysThreadBootstrap( Thrd **thrd );
```

This routine turns the executing Unix process into the initial thread, returning the thread ID of the new thread. Under JN, this call simply returns the ID of the executing thread.

returns: return code, sets `thrd` to the thread ID of the new thread.

return val: `SYS_OK` – Completed normally.

```
void sysThreadInitializeSystemThreads();
```

The internal threads used by the threading package are initialized. A clock thread, idle thread, ‘finalization’ thread, and garbage collection thread are created. Under JN, this call is not needed for correct execution. It is included only to satisfy the reference of the Java Interpreter.

returns: none.

```
int sysThreadSingle();
```

Threads can run exclusive by starting a critical section with `sysThreadSingle()` and ending the critical section with `sysThreadMulti()`, that is, this routine disables active multithreading. Code that calls this routine should always terminate the resulting critical section with a `sysThreadMulti()` call.

This routine works by disabling the scheduler. Interrupts are still enabled. No thread other than the invoking thread will be run, not even the null thread. The invoking thread can continue to issue all JN API calls. If there are no runnable tasks, the kernel waits for an event to ready the invoking thread.

returns: This call always returns `SYS_OK`.

```
void sysThreadMulti();
```

Threads can run exclusive by starting a critical section with `sysThreadSingle()` and ending the critical section with `sysThreadMulti()`, that is, this routine resumes active multithreading.

returns: There are no return values. It is a system error if this call does not match a preceding `sysThreadSingle()` call.

```
int sysThreadCreate( long          stack_size,
                    unsigned int  flags,
                    void          *(*start)(void *),
                    Thrd          **thrd,
                    void          *argument );
```

Creates a suspended new thread with a stack of the indicated size. The thread ID is returned via argument `thrd`. Thread execution will begin at address `start`, which should be the address of a C subroutine. A `sysThreadCreate()` call should be followed by a `sysThreadResume()` when the caller wishes to activate the newly created thread.

The single argument is passed on the new thread’s call stack to the routine at address `start`.

All thread’s contain a *cookie*. The *cookie* is an arbitrary pointer stored in the thread’s context. It can be set and obtained by `sysThreadSetBackPtr()` and `sysThreadGetBackPtr()`.

New threads always start executing at `NORM_PRIORITY`, that is, priority 5. The child's priority can be altered by `sysThreadSetPriority()`. Note that the parent of a thread can alter the priority of a newly created child before the child ever executes, as `sysThreadCreate()` does not block the parent. If the parent sets the priority of a child higher than the parent itself, the child is eligible to run before the `sysThreadSetPriority()` in the parent returns.

The only supported `flags` value is `THR_USER`, which indicates that this is not a system thread.

return values:

- `SYS_ERR` – Couldn't do it.
- `SYS_OK` – Normal completion.

```
void sysThreadExit();
```

This thread is called automatically when the initial thread function returns (i.e., the routine specified in the `sysThreadCreate()` 'returns' to a call of this function. This function frees all thread resources and terminates the thread. There are no return values and no errors returned).

Monitors that are owned by the thread are not automatically freed. Routine `sysMonitorDestroy()` can be called to force a given thread to release a given monitor.

This routine can be called directly, although the recommended means of terminating from a thread is to return from the top level, that is, from the routine specified in the `sysThreadCreate()`.

There is no way to force termination of an arbitrary thread via the API.

returns: none.

```
Thrd *sysThreadSelf();
```

Returns the *thread ID* of the executing thread. JN thread IDs are simply pointers to the internal thread data structure, which is a potential security risk.

returns: The thread ID is the only return value.

```
void sysThreadYield();
```

This call simply yields the processor. The running thread that makes this call goes 'to the end of the line' behind other threads at the same priority level that are ready to execute, if any exist. If there are no other runnable threads, the current thread continues executing.

This call implements non-preemptive 'round-robin' scheduling.

The thread is only rotated to the 'end-of-the-line' with respect to threads at its current priority level.

returns: There are no return values from this call.

```
int sysThreadSuspend( Thrd *thrd );
```

This call suspends the indicated thread, which may be that of the caller. If the indicated thread exists, it is placed in a `SUSPENDED` state where it is never eligible for execution.

A `sysThreadResume()` call must be made to resume execution of the suspended thread. If the target thread is the caller itself, the return from `sysThreadSuspend()` will not occur until after a `sysThreadResume()` has reactivated the thread.

return values:

- `SYS_ERR` – It couldn't be done.
 - `SYS_OK` – Success.
-

```
int sysThreadResume( Thrd *thrd );
```

The indicated thread, which should be `SUSPENDED`, is resumed, that is, it is made eligible for execution. It is not an error if the thread is not suspended - the call is simply ignored.

A thread is suspended by calling `sysThreadSuspend()`.

return values: `SYS_OK` – Success.

```
int sysThreadSetPriority( Thrd *thrd,
                        int priority );
```

The priority of the indicated thread is changed. The target thread need not be the caller. If the target thread is the caller, the effect of the priority change occurs immediately, which may result in the caller losing control of the processor.

To be compatible with Java, `MIN_PRIORITY` is defined as 1, `MAX_PRIORITY` is defined as 10, and `NORM_PRIORITY` is defined as 5. The highest priority thread is selected for execution. New threads are created initially at `NORM_PRIORITY`, that is, at priority 5.

Errors are considered fatal.

return values: `SYS_OK` – Success.

```
int sysThreadGetPriority( Thrd *thrd,
                        int *priority );
```

The priority of the indicated thread is returned via the `priority` argument.

Errors are considered fatal.

return values:

`SYS_OK` – Success.

```
void *sysThreadGetBackPtr( Thrd *thrd );
```

The *cookie* argument stored in the indicated thread's context by a `sysThreadSetBackPtr()` call is returned.

returns: The cookie value. There are no status return values.

```
void sysThreadSetBackPtr( Thrd *thrd,  
                          void *new_cookie );
```

The *cookie* field in the indicated thread's context is set to the *new_cookie* argument.
returns: There is no return status. A bad thread pointer is considered a fatal error.

```
int sysThreadCheckStack();
```

This function returns a 1 if the amount of free space in the caller's stack is greater than JN *sysgen* manifest constant *STACK_REDZONE*, otherwise it returns a 0. The default JN *STACK_REDZONE* is 4K.

returns:

- 0 - No stack space left.
 - 1 - Stackspace is left.
-

```
void sysThreadPostException( Thrd *thrd,  
                             void *exception );
```

This call posts an exception to a thread, that is, it triggers an exception handler to run in thread's context. It is not clear that Java has defined a standard portable method for dealing with this yet....

returns: none.

```
void *sysThreadStackBase( Thrd *thrd );
```

This call returns the base address of the stack for the indicated thread, that is, the address from which the stack grows *down*.

returns: The top stack address. There are no status return values. A bad thread address is considered a fatal error.

```
void *sysThreadStackPointer( Thrd *thrd );
```

This call returns the current stack pointer of the indicated thread, which can be that of the caller.

returns: The stack pointer. There are no status return values. A bad thread address is considered a fatal error.

```
int sysThreadEnumerateOver( int (*func)( Thrd *, void *),  
                           void *arg );
```

This routine provides an iterator that applies a function to all threads. For each existing thread, the application-supplied user function indicated by argument *func* is called. The user function is supplied 2 arguments, the address of the thread and the pass-through argument *arg*.

The address of the thread is the thread ID, so the user function receives a different thread ID every time it is called.

In addition to the application-level context, the *arg* argument can be used to specify arbitrary arguments to the user-level routine. This pointer can be used to point to whatever data, data structures, or command blocks that the user desires.

returns:

If the application-supplied function (that is, the function supplied by the caller) does not return `SYS_OK`, the enumeration stops. If this occurs, `sysThreadEnumerateOver()` returns the return code generated by the user function. If all calls to the user function return `SYS_OK`, `sysThreadEnumerateOver()` returns `SYS_OK`.

```
void sysThreadDumpInfo( Thrd *thrd );
```

In JN this routine produces a dump of the thread control blocks.

3 Monitor Calls

Although *monitors* are perhaps the most ubiquitous modern concurrent programming construct, monitor details often vary. In this section, a *monitor* can be considered a critical section associated with two semaphores. One semaphore guards *external* entrance into the critical section, and the other, which starts with a value of 0 (unavailable), is used to guard *internal* access. To enter a critical section, a thread must use the external semaphore and `sysMonitorEnter()`. Only one thread can be inside a critical section at a time. Once inside the critical section, if the active thread must wait for some occurrence (an I/O completion or a change in the content of a data structure, etc.), it uses `sysMonitorWait()` to put itself on the *internal* semaphore queue, while releasing its hold, atomically, on the external semaphore. Thus an external thread that was forced to wait can enter the critical section. Whenever a thread uses `sysMonitorExit()` to leave the critical section, as with any semaphore operation, another thread waiting on the external semaphore proceeds into the critical section.

Threads waiting on the internal semaphore are only reactivated by an explicit `sysNotify()` or `sysNotifyAll()` operation. Essentially this is a semaphore $V()$ operation on the internal semaphore. An active routine that completes an activity upon which some thread may be internally waiting issues these calls. These calls simply move the thread waiting internally to the end of the external wait queue. When each waiting thread moves to the head of the external wait queue, its execution resumes at the point inside the critical section where it waited for the needed resource or event.

A `sysNotify()` or `sysNotifyAll()` can only be performed by a thread that is inside the critical section. Race conditions in which `sysNotify()` is called before what should be the corresponding `sysMonitorWait()` can thus be avoided by correct programming. However, a newly activated thread that has become unblocked should always recheck the condition on which it was waiting. Such a thread cannot tell if a `sysNotify()` or `sysNotifyAll()` activated it, and in the case of `sysNotifyAll()` the resource may not be available by the time the thread actually resumes running in the critical section.

```
int sysMonitorInit( Monitor *sem,
                  bool_t in_cache );
```

If the `in_cache` flag is non-zero, the semaphore flags are marked `SYS_MON_IN_CACHE`, indicating that this is a *dynamic* monitor that will be deleted when no longer needed. The `sem` argument points to a JN Sem structure that is to be initialized. This structure is allocated by the application. The application should be careful not to allocate this structure as an automatic on the C stack and then continue to use it after returning from the function that allocated it.

This routine does not invoke the JN kernel; it simply performs data structure initialization.

returns: This function always returns `SYS_OK`;

```
int sysMonitorEnter( Monitor *sem );
```

If the critical section guarded by the semaphore indicated by `sem` is not in use, this call lets the calling thread enter the critical section. Otherwise, the caller is blocked and queued on the external waiting queue of the indicated semaphore. This call must always be followed by a `sysMonitorExit()` at the end of the critical section protected by the semaphore.

A bad `sem` address is considered a fatal error.

returns: This function always returns `SYS_OK`. This function only returns when the caller is allowed to proceed within the critical section.

```
int sysMonitorExit( Monitor *sem );
```

This routine is called to exit a critical section controlled by the indicated semaphore. The caller must have previously acquired the semaphore via `sysMonitorEnter()`. If any threads are blocked on the external wait queue of the semaphore, one will be selected to proceed when the caller leaves the critical section. Any threads waiting on the internal wait queue of the semaphore are unaffected.

If no threads are waiting on the semaphore, and the semaphore is marked `SYS_MON_IN_CACHE`, this call returns with `SYS_DESTROY`, indicating to higher-level routines that the semaphore data structure can be deallocated if need be. To safely use monitors in such a fashion, either a safe programming convention can be used which assures there can be no race condition (the monitor is only deleted when the last thread using it receives a `SYS_DESTROY`), or a *static* monitor can guard entrance, exit, allocation, and deallocation of the code guarded by one or more dynamic monitors.

returns:

- `SYS_ERR` – The caller does not own the indicated semaphore.
- `SYS_DESTROY` – The caller successfully exited the critical section, the `in_cache` flag was non-zero on the original `sysMonitorInit()` call, and no other thread was unblocked to enter the critical section.
- `SYS_OK` – The caller successfully exited the critical section and another thread was unblocked to enter the critical section.

```
int      sysMonitorDestroy( Monitor *sem,
                          Thrd   *thrd );
```

This routine is used when deleting a thread. This call does not deallocate the semaphore. Rather, it can be considered a forced `sysMonitorExit()` on a thread with respect to a given semaphore.

If the specified thread owns the indicated semaphore, that is, is inside the critical section and not waiting, the effect of this call is as if `sysMonitorExit()` had been called by the specified thread. The thread releases control of the semaphore. If no other threads exist on any of the semaphore's wait queues, `SYS_DESTROY` is returned, potentially indicating that semaphore usage is complete and that the application can deallocate the semaphore. If other threads exist on the semaphore's wait queues, `SYS_OK` is returned, and the head of the external wait queue is unblocked to enter the critical section.

If the specified thread does not own the indicated semaphore, this call has no effect. Presumably, this is because a thread never is terminated while in a wait state.

returns:

- `SYS_OK` – If the caller does not own the indicated semaphore, this call has no effect. If the caller owns the semaphore, the `sysMonitorDestroy()` failed in the sense that the semaphore cannot be deleted – it performed a `sysMonitorExit()` function instead, releasing another thread to enter the critical section.
- `SYS_DESTROY` – No threads are waiting on the semaphore, it can be removed.

```
bool_t  sysMonitorEntered( Monitor *sem );
```

This function returns `True (1)` if the caller owns the semaphore (is currently in the critical section). The return type is defined as an integer flag.

returns:

- 1 – in the critical section controlled by `sem`.
- 0 – not in the critical section controlled by `sem`.

```
int      sysMonitorWait( Monitor *sem,
                       int      millis );
```

A thread inside a critical section uses this call to block and await either an event or the specified number of milliseconds. The caller waits on the semaphore's internal wait queue. After the `sysMonitorWait()` call, the event is triggered by a `sysNotify()` or `sysNotifyAll()` call. Such a call is issued by some other active thread, which owns the semaphore at the time it performs the notify.

When activated by either a notification event or the passage of the indicated time interval, the thread is placed on the semaphore's *external* wait queue. This queue contains threads waiting to run, one at a time, in the critical section.

The `sysMonitorWait()` call places the calling thread on an the internal wait queue associated with `sem`. The thread must have already entered the critical section via `sysMonitorEnter()`. Typical reasons to use wait include awaiting I/O completion, waiting for data to be placed in an input buffer, and so on.

If the `millis` argument is specified as `SYS_TIMEOUT_INFINITY`, there is no timeout associated with the wait.

The internal wait queue is not a counted semaphore, thus a `sysNotify()` call or event completion that precedes the `sysMonitorWait()` has no effect. For this reason, and also because a `sysNotifyAll()` unblocks all threads waiting on the semaphore's internal queue, code that performs a `sysMonitorWait()` should not assume that it has been correctly unblocked. Rather, it should always explicitly check that the condition on which it has waited has actually occurred, and if it has not, it should reissue the `sysMonitorWait()` call.

returns:

- `SYS_ERR` – The caller must own the indicated semaphore.
- `SYS_OK` - Normal completion, which indicates that the wait has completed. Either the event has occurred or the specified time interval has passed.

```
int sysMonitorNotify( Monitor *sem );
```

The thread at the head of the semaphore's internal wait queue is put on the semaphore's external wait queue. Each semaphore has both an external and internal wait queue. The external queue contains threads waiting to run in the critical section controlled by the semaphore. The internal queue is used by threads which, while they were inside the critical section, needed to block awaiting either an event or passage of a particular time interval.

`sysMonitorNotify()` must be called by code that is inside the critical section. It is common, for instance, for code that entered the critical section and wrote some data into a data structure, to call `sysMonitorNotify()` before it calls `sysMonitorExit()`. Thus, a thread that entered the critical section to read data from the data structure, but found none and thus called `sysMonitorWait()`, will be unblocked and can proceed.

returns:

- `SYS_OK` – Normal completion.
- `SYS_ERR` – The caller does not own the semaphore.

```
int sysMonitorNotifyAll( Monitor *sem );
```

All threads waiting on a semaphore's internal wait queue are moved to the semaphore's external wait queue. See `sysMonitorNotify()`. This call is identical to `sysMonitorNotify()` except that all threads on the internal wait queue are unblocked. Each unblocked thread, as it 'awakes' within the critical section, must recheck conditions to see if it can proceed or if it should issue another `sysMonitorWait()`.

returns:

- `SYS_OK` – Normal completion.
- `SYS_ERR` – The caller does not own the semaphore.

```
int sysMonitorSizeof();
```

This routine is simply a cover function for `sizeof(Monitor)`. Since semaphore data structures are allocated at the user level, this call is used so that high-level routines can determine the size of the data structure they must allocate.

```
void sysMonitorDumpInfo( Monitor *sem );
```

This is a debug routine that dumps the owner of a semaphore and the threads on the semaphore's wait queues.

4 File Calls

The *sys_api.h* porting interface describes a set of file management functions that map almost directly into standard Unix file APIs. There are 2 sets of routines that are identical except that in one case the name is followed with `FD`, that is, `sysOpen()` becomes `sysOpenFD()`. Additionally, `sysOpenFD()` has an additional first argument, which is the address of a 32-bit int that is to receive the file handle.

JN files are simply in-memory queues (RAM files). These queues share Unix file semantics. Each `sysOpen()` returns a unique file handle that has a unique position within the file. Files are simply byte-streams. Arbitrary byte substrings can be read from and written to the file.

JN files need not be contiguous, that is, Unix sparse file semantics are supported.

JN files are implemented as queues of *segments*. A segment is a buffer descriptor. Although most segment buffers are allocated from a fixed array, buffer segments can be variable length and can thus be used to describe *preloaded* files that are linked into a single buffer in the system image.

All file I/O is currently synchronous – it is just a buffer copy to or from the appropriate location in the queue.

```
void sysInitFD( int *fdptr,
               int descr );
```

This call provides a way for a thread to support Unix style `stdin`, `stdout`, and `stderr` file handles. Normally JN generates an arbitrary non-zero 32-bit file handle. However, these 3 `std` handles, by Unix convention, always have the values of 0, 1, and 2.

A JN thread can call this function 3 times, with `descr` arguments of 0, 1, and 2, and get back file handles that correspond to standard input, output, and error.

returns: File descriptor `fdptr` is set. There are no status return values.

```
int sysMode_Line( int *fdptr );
```

The specified handle is set to *line mode*. This is only useful for the serial (uart) driver. The driver will read and accumulate characters until a carriage return (C '\r', 0x0D) is encountered. A read request can thus read an entire line of characters. Likewise, an entire buffer can be written in a single write request.

In line mode, a terminating carriage return is converted to a line feed (C '\n', 0x0A).

returns: Zero is returned on success.

```
int sysMode_Byte( int *fdptr );
```

The specified handle is set to *byte mode*. This is only useful for the serial (uart) driver. In byte mode, reads complete when a single character is available, and the driver does character processing - input characters are returned to the application exactly as received, including XON-XOFF flow control characters. Likewise, application output characters are not altered in any way by the driver.

returns: Zero is returned on success.

```
int sysOpen( char *fname,
             int flags,
             int mode );

int sysOpenFD( int *fdptr,
              char *fname,
              int flags,
              int mode );
```

Open the file identified by `fname` using the specified `flags` and `mode`. The file handle for the new file is returned. In the case of `sysOpenFD()` the file handle is also returned in the 32-bit location specified by the first argument. This returned file handle will never be null.

JN file descriptor structures consist only of a single integer which contains the file handle.

The only flag currently supported is `O_CREAT`, which causes a new file to be created.

Note this call can create and overwrite files.

returns:

- Upon success the file handle is returned. The file handle is an integer greater or equal to 0.
- -1 – This is the value of `SYS_ERR`, and is returned on error.

```
int sysClose( int *fdptr );
int sysCloseFD( int *fdptr );
```

The specified file is closed.

Note that if real async I/O is supported, multiple readers may be in the process of reading, so the file is simply marked as closing, and the file is actually closed when the file descriptor usage count falls to zero at the end of an I/O.

returns:

- `SYS_ERR` on error.
- `SYS_OK` on success.

```
int sysRead( int *fdptr,
             char *buf,
             int nbytes );
int sysReadFD( int *fdptr,
              char *buf,
              int nbytes );
```

The given number of bytes, `nbytes`, are read from the file designated by the handle in `fdptr` into user buffer `buf`. The bytes are read starting at the location at which the file handle is initially located. The current file position of the handle is set to the location one byte past the last byte read.

returns:

- The number of bytes read are returned on success.
- `SYS_ERR - (-1)` is returned on failure.

```
int sysWrite( int *fdptr,
             char *buf,
             int nbytes );
int sysWriteFD( int *fdptr,
               char *buf,
               int nbytes );
```

The given number of bytes, `nbytes`, are written from the user buffer `buf` to the file designated by `fdptr`. The bytes are written to the location at which the file handle is initially located. At the end of the write, the handle's current file position is set to the location one byte past the last byte written.

returns:

- The number of bytes written are returned on success.
- `SYS_ERR - (-1)` is returned on failure.

```
int    sysLseek(    int    *fdptr,
                   long    offset,
                   long    whence );

int    sysLseekFD( int    *fdptr,
                  long    offset,
                  long    whence );
```

The current file position in the indicated file is set to the location specified by the `offset` argument. The `whence` argument indicates the interpretation of the offset:

- `SEEK_SET` - The offset is absolute, that is, the exact file address.
- `SEEK_CUR` - The offset is relative to the current file position.
- `SEEK_END` - The offset is relative to the end of the file.

The `sysLseek()` call can set the file position to beyond the current end of file, and to a negative file location. This is not considered an error.

returns:

The new file location. A `SYS_ERR (-1)` is returned if the file descriptor is not valid.

```
int    sysAvailable( int    *fdptr,
                    long    *pbytes );

int    sysAvailableFD( int    *fdptr,
                      long    *pbytes );
```

The number of bytes which remain in the file between the current file position and the end of the file are returned via argument `pbytes`.

returns:

A 0 is returned on any failure. A 1 is returned on success.

NOTE!! — These return values are not consistent with other return code usage. `SYS_OK` is defined as 0, and `SYS_TIMEOUT` as 1. Note that 0 is returned on `_failure_`.

5 Exception/Signal Calls

JN supports software interrupts. As with a real hardware interrupt, a software interrupt is a routine that is to be run whenever some condition occurs. Each thread can establish its own set of software interrupt handlers.

When the system detects an exception, that is, a software interrupt condition such as a divide-by-zero, if the corresponding thread has a handler for the exception, JN queues a software interrupt notification to the thread. The handler will execute as soon as the thread becomes the highest priority executable thread.

As with real interrupts, software interrupt handlers run on the stack below the normal thread stack pointer. A handler can execute all JN APIs. A handler is simply a normal C subroutine, externally indistinguishable from any other subroutine used by the thread, that is, handler code is identical to 'normal' code. The programmer should keep in mind, however, that the handler may run at any point in the execution of the thread, unless exception handling is explicitly disabled as described in the following section.

```
int  sys_set_exception( int    exception_number,
                       void   (*handler)(void) )
```

A routine in the calling thread, specified by argument `handler`, will be called when the exception indicated by `exception_number` occurs. The exception handler remains in effect for all occurrences of the specified exception with respect to the calling thread, until `sys_set_exception()` is called with the `handler` argument set to 0.

JN exception numbers are arbitrary integers. Currently, one is predefined in `jn.h`, `EXP_DVZ` (divide by zero).

```
int  sysInterruptsPending();
```

Returns True (1) if the invoking thread has pending software interrupts. A thread may have pending software interrupts since software interrupts are delivered one-at-a-time to the thread and execute to completion. In addition, the thread can disable delivery of software interrupts by calling `intrLock()`, for instance, if it is updating a memory resident data structure that an exception handler will also update.

```
void  intrLock(void);
```

This routine disables all software interrupt delivery to the invoking thread. Software interrupts are still queued to the thread, however, their delivery is postponed until an `intrUnlock()` call occurs. An `intrLock()` call should always be followed by an `intrUnlock()` call.

The `intrLock()` call is typically used when a thread is going to perform an operation, such as executing a critical section, in which delivering a software interrupt could cause some concurrency problem. In this case, `intrLock()` is used to defer software interrupts until the `intrUnlock()` issued after exiting the critical section.

```
void  intrUnlock(void);
```

Enable all software interrupts. This routine is called after deferring software interrupt delivery by a call to `intrLock()`. Any pending software interrupts will be delivered to the thread that issues this call before the call returns. Since software interrupts can queue to the thread, a given thread's software interrupt handler may execute more than once. For instance, a `SIG_ALARM` handler may have a number of `ALARM` interrupts to handle.

returns: None, however, if any pending software interrupts exist, this call will not return until all software interrupts have been handled, that is, after the thread's handler routines have run and processed all pending software interrupts.

6 Standard Unix Socket Programming Functions

The following routines provide a standard C socket programming interface. These routines are the API to the KA9Q networking code. These routines have been made as equivalent to the corresponding Unix calls as possible. You should be able to use Unix man to obtain descriptions of all the following routines.

```
long          htonl( long input )
unsigned int  ntohl( unsigned int i_val)

int          accept(      int s,struct sockaddr *peername,int *peernameelen );
int          bind(       int s,struct sockaddr *name,int namelen );
int          close_s(    int s);
int          connect(    int s,struct sockaddr *peername,int peernameelen );

int          getpeername( int s,struct sockaddr *peername,int *peernameelen );
int          getsockname( int s,struct sockaddr *name,int *namelen );
int          listen(     int s,int backlog );

int          sock_shutdown( int s,int how );
int          socket(      int af,int type,int protocol );

int          sockkick(   int s );
int          socklen(    int s,int rtx );

int          socketpair( int af,int type,int protocol,int sv[] );

int          recv(       int s,void *buf,int len,int flags );
int          recvfrom(   int s,void *buf,int len,int flags,
                        struct sockaddr *from,int *fromlen );

int          send(       int s,void *buf,int len,int flags );
int          sendto(     int s,void *buf,int len,int flags,
                        struct sockaddr *to,int tolen );
```

7 Miscellaneous ‘Standard’ Unix Routines

The following routines are simple equivalents of the corresponding Unix functions:

```
char *jn_malloc( int num_bytes_needed )
char *sysMalloc( int num_bytes_needed )

void jn_free( char *p )
void sysFree( char *p )

char *jn_realloc( char *p,
                 int num_bytes )
void *sysRealloc( void *p,
                 size_t num_bytes )

char *sysCalloc( int num_elements,
                int num_bytes )

void perror( char *msg )

void printf( const char *fmt, ... )
void fprintf( int handle,
              const char *fmt, ... )

int fstat( int handle,
           struct stat* st )
int sysStat( char *path,
             struct stat *s )

int sysMkdir( char *path,
              int mode )

int sysUnlink( char *path )

char *strdup( char *p )

int sysPutenv( char *env_string )
char *sysGetenv( char *name_string )
```

These routines are all intended to work exactly as their Unix counterparts, that is, you can determine their functionality by doing a Unix man for the equivalent Unix routine. In some cases, such as `strdup()`, JN provides the routine simply because it was not included in one of the embedded development environments that we were using.

The `sysMkdir()` routine does nothing and always returns success. File names in JN can be up to 128 characters long and can contain Unix style directory separators (that is, the `"/"`). Thus, there is no need in JN to create a directory before populating it with files.

The environment string routines `sysPutenv()` and `sysGetenv()` work as in Unix. The string input to `sysPutenv()` has a "name=value" format. A `sysGetenv()` routine can later supply the name and will retrieve the associated value. There is only one global environmental string table.

Routines `fstat()` and `sysStat()` can be used to obtain file sizes.

Routine `perror()` prints a message and crashes the system.

8 Time Functions

Getting time routines correct is probably one of the harder things in system software. Doing time really right can be difficult because there are a number of special cases (does your city *not* observe daylight savings time, even though your state does?). Are leap years and leap seconds handled OK? Even parsing and formatting time and dates can be interesting (see `strptime()`). To see how complex this can become, examine the `zic` command on OSF Unix machines, which is basically an expert system compiler.

JN currently provides minimal time support. The JN `gettimeofday()` API routine is basically 'Berkeley compatible'. The time routines used in JN were originally, long ago (years ago), based on an astronomy handbook with code in Basic or a calculator language; I forget the details. The code is almost a verbatim translation. I believe the author was French or Belgian.

In the following, Unix time (seconds since 00:00 1-Jan-1970 at Greenwich (GMT)) is called *epoch time*. The Julian day is the number of days since 12:00 (noon) 1-Jan-4713 BC at Greenwich.

```
gettimeofday( struct timeval  *tv,
              struct timezone *tz )
```

As in Unix, the current system time is returned in the `timeval` structure as 2 32-bit values, with `tv_sec` containing the current Unix epoch seconds (seconds since 00:00 01-Jan-1900 GMT), and `tv_usec` containing the number of milliseconds since the last epoch second.

WARNING: the values returned in the `timezone` structure are currently hard-coded to UCSC, that is, `tv_minuteswest` is set to 8 (we are 8 hours west of Greenwich). The daylight savings flag, `tz_dsttime`, is set to 0.

```
time_t  jn_gmt_tm_to_time_t( struct tm *tm )
```

The Gregorian date/time (normal wall calendar date/time) that is stored in the `tm` structure is converted into Unix epoch seconds (seconds since 00:00 01-Jan-1900 GMT). This 32-bit `time_t` value is returned.

```
long  jn_gmt_dmyhms_to_epoch.UTC( int  day,
                                  int  month,
                                  int  year,
                                  int  hour,
                                  int  minute,
                                  int  second )
```

A Gregorian (wall calendar) date/time is input. Output is the corresponding Unix epoch second (seconds since 00:00 01-Jan-1900 GMT).

This routine is the same as `jn_gmt_tm_to_time_t()`, except that the input arguments do not have to be marshaled into a `struct tm`.

```
jn_julian_day_to_gregorian_dmy(
    long  jul_day, /* Convert julian day to equivalent */
    int  *day,    /* 1 origin day of month (1-31), */
    int  *month,  /* 1 origin month (1=jan), and */
    int  *year ) /* integer year (e.g., 1996). */
```

A Julian day is input as the first argument. The corresponding Gregorian (wall calendar) day, month, and year are returned. Note this is the real Julian day, not the day of the year, which is sometimes confusingly called the Julian day or date.

```
long jn_gregorian_dmy_to_julian_day(  
  
    int day, /* Given the year (e.g., 1996) and */  
    int month, /* the 1 origin month (jan==1), and */  
    int year ) /* 1 origin day (1-31), calculate */  
                /* the julian day. */
```

A Gregorian (wall calendar) day, month, and year are supplied as input. Output is the corresponding Julian day.

9 Debugging Support

There are a number of internal debugging routines that have been exposed at various times to external applications. Currently `sysCollectChecksum()` is the only such routine.

```
void    sysCollectChecksum( int i,  
                           char *who )
```

This routine checksums all the compiled code in JN. Since JN is used on systems without memory management, this is useful in detecting bugs that corrupt code. This routine will print a message if the code checksum changes. The arguments are arbitrary and identify the caller; they will be echoed on the display.

10 Unimplemented *sys_api.h* Functions

File *sys_api.h* specifies an external API. If this API exists, the JVM can run. However, there are some *sys_api.h* APIs that simply do not make sense in an embedded environment. These routines have not been implemented. In most cases they have been stubbed, that is, there is an implementation that simply crashes the system after printing an error message.

The unimplemented routines are:

```
sysInitializeLinker()  
sysAddDLLSegment()  
sysSaveLDPath()  
sysDynamicLink()  
sysBuildName()  
  
sysMapMem()  
sysUnmapMem()  
sysCommitMem()  
sysUncommitMem()  
  
sysExit()  
sysAtexit()  
sysAbort()  
  
sysGetMilliTicks()  
sysTime()  
sysTimeMillis()  
sysLocaltime()  
sysGmtime()  
sysStrftime()  
sysMktime()
```

Dynamic linking and virtual memory operations are not supported by JN (and many other embedded systems). Routine `sysExit()` would correspond to an orderly system shutdown. JN has no 'shutdown' functionality.