THE VISOPSYS KERNEL API (Version 0.6)

All of the kernel's functions are defined in the file /system/headers/sys/api.h. In future, this file may be split into smaller chunks, by functional area. Data structures referred to in these function definitions are found in header files in the /system/headers/sys directory. For example, a 'disk' object is defined in /system/headers/sys/disk.h.

One note on the 'objectKey' type used by many of these functions: This is used to refer to data structures in kernel memory that are not accessible (in a practical sense) to external programs. Yes, it's a pointer -- A pointer to a structure that is probably defined in one of the kernel header files. You could try to use it as more than just a reference key, but you would do so at your own risk.

All functions divided by functional area:

Text input/output functions

`objectKey textGetConsoleInput(void)`

Returns a reference to the console input stream. This is where keyboard input goes by default.

`int textSetConsoleInput(objectKey newStream)`

Changes the console input stream. GUI programs can use this function to redirect input to a text area or text field, for example.

`objectKey textGetConsoleOutput(void)`

Returns a reference to the console output stream. This is where kernel logging output goes by default.

`int textSetConsoleOutput(objectKey newStream)`

Changes the console output stream. GUI programs can use this function to redirect output to a text area or text field, for example.

`objectKey textGetCurrentInput(void)`

Returns a reference to the input stream of the current process. This is where standard input (for example, from a getc() call) is received.

`int textSetCurrentInput(objectKey newStream)`

Changes the current input stream. GUI programs can use this function to redirect input to a text area or text field, for example.

`objectKey textGetCurrentOutput(void)`

Returns a reference to the console output stream.

`int textSetCurrentOutput(objectKey newStream)`

Changes the current output stream. This is where standard output (for example, from a putc() call) goes.

`int textGetForeground(void)`

Get the current foreground color as an int value. Currently this is only applicable in text mode, and the color value should be treated as a PC built-in color value. Here is a listing: 0=Black, 4=Red, 8=Dark gray, 12=Light red, 1=Blue, 5=Magenta, 9=Light blue, 13=Light magenta, 2=Green, 6=Brown, 10=Light green, 14=Yellow, 3=Cyan, 7=Light gray, 11=Light cyan, 15=White

`int textSetForeground(int foreground)`

Set the current foreground color from an int value. Currently this is only applicable in text mode, and the color value should be treated as a PC builtin color value. See chart above.

`int textGetBackground(void)`

Get the current background color as an int value. Currently this is only applicable in text mode, and the color value should be treated as a PC builtin color value. See chart above.

`int textSetBackground(int background)`

Set the current foreground color from an int value. Currently this is only applicable in text mode, and the color value should be treated as a PC builtin color value. See chart above.

`int textPutc(int ascii)`

Print a single character

`int textPrint(const char *str)`

Print a string

`int textPrintLine(const char *str)`

Print a string with a newline at the end

`void textNewline(void)`

Print a newline

`int textBackSpace(void)`

Backspace the cursor, deleting any character there

`int textTab(void)`

Print a tab

`int textCursorUp(void)`

Move the cursor up one row. Doesn't affect any characters there.

`int textCursorDown(void)`

Move the cursor down one row. Doesn't affect any characters there.

`int textCursorLeft(void)`

Move the cursor left one column. Doesn't affect any characters there.

`int textCursorRight(void)`

Move the cursor right one column. Doesn't affect any characters there.

`void textScroll(int upDown)`

Scroll the current text area up (-1) or down (+1)

`int textGetNumColumns(void)`

Get the total number of columns in the text area.

`int textGetNumRows(void)`

Get the total number of rows in the text area.

`int textGetColumn(void)`

Get the number of the current column. Zero-based.

`void textSetColumn(int c)`

Set the number of the current column. Zero-based. Doesn't affect any characters there.

`int textGetRow(void)`

Get the number of the current row. Zero-based.

`void textSetRow(int r)`

Set the number of the current row. Zero-based. Doesn't affect any characters there.

`void textSetCursor(int on)`

Turn the cursor on (1) or off (0)

`int textScreenClear(void)`

Erase all characters in the text area and set the row and column to (0, 0)

`int textScreenSave(void)`

Save the current screen in an internal buffer. Use with the textScreenRestore function.

`int textScreenRestore(void)`

Restore the screen previously saved with the textScreenSave function

`int textInputStreamCount(objectKey strm)`

Get the number of characters currently waiting in the specified input stream

`int textInputCount(void)`

Get the number of characters currently waiting in the current input stream

`int textInputStreamGetc(objectKey strm, char *cp)`

Get one character from the specified input stream (as an integer value).

`int textInputGetc(char *cp)`

Get one character from the default input stream (as an integer value).

`int textInputStreamReadN(objectKey strm, int num, char *buff)`

Read up to 'num' characters from the specified input stream into 'buff' `int textInputReadN(int num, char *buff)`

Read up to 'num' characters from the default input stream into 'buff'

`int textInputStreamReadAll(objectKey strm, char *buff)`

Read all of the characters from the specified input stream into 'buff'

`int textInputReadAll(char *buff)`

Read all of the characters from the default input stream into 'buff'

`int textInputStreamAppend(objectKey strm, int ascii)`

Append a character (as an integer value) to the end of the specified input stream.

`int textInputAppend(int ascii)`

Append a character (as an integer value) to the end of the default input stream.

`int textInputStreamAppendN(objectKey strm, int num, char *str)`

Append 'num' characters to the end of the specified input stream from 'str'

`int textInputAppendN(int num, char *str)`

Append 'num' characters to the end of the default input stream from 'str'

`int textInputStreamRemove(objectKey strm)`

Remove one character from the start of the specified input stream.

`int textInputRemove(void)`

Remove one character from the start of the default input stream.

`int textInputStreamRemoveN(objectKey strm, int num)`

Remove 'num' characters from the start of the specified input stream.

`int textInputRemoveN(int num)`

Remove 'num' characters from the start of the default input stream.

`int textInputStreamRemoveAll(objectKey strm)`

Empty the specified input stream.

`int textInputRemoveAll(void)`

Empty the default input stream.

`void textInputStreamSetEcho(objectKey strm, int onOff)`

Set echo on (1) or off (0) for the specified input stream. When on, any characters typed will be automatically printed to the text area. When off, they won't.

`void textInputSetEcho(int onOff)`

Set echo on (1) or off (0) for the default input stream. When on, any characters typed will be automatically printed to the text area. When off, they won't.

Disk functions

`int diskReadPartitions(void)`

Tells the kernel to (re)read the disk partition tables.

`int diskSync(void)`

Tells the kernel to synchronize all the disks, flushing any output.

`int diskGetBoot(char *name)`

Get the disk name of the boot device. Normally this will contain the root filesystem.

`int diskGetCount(void)`

Get the number of logical disk volumes recognized by the system

`int diskGetPhysicalCount(void)`

Get the number of physical disk devices recognized by the system

`int diskGet(const char *name, disk *userDisk)`

Given a disk name string 'name', fill in the corresponding user space disk structure 'userDisk.

`int diskGetAll(disk *userDiskArray, unsigned buffSize)`

Return user space disk structures in 'userDiskArray' for each logical disk, up to 'buffSize' bytes.

`int diskGetAllPhysical(disk *userDiskArray, unsigned buffSize)`

Return user space disk structures in 'userDiskArray' for each physical disk, up to 'buffSize' bytes.

`int diskGetPartType(int code, partitionType *p)`

Gets the partition type data for the corresponding code. This function was added specifically by use by programs such as 'fdisk' to get descriptions of different types known to the kernel.

```
partitionType *diskGetPartTypes(void)
```

Like diskGetPartType(), but returns a pointer to a list of all known types.

```
int diskSetLockState(const char *name, int state)
```

Set the locked state of the disk 'name' to either unlocked (0) or locked (1)

```
int diskSetDoorState(const char *name, int state)
```

Open (1) or close (0) the disk 'name'. May require a unlocking the door first, see diskSetLockState().

```
int diskGetMediaState(const char *diskName)
```

Returns 1 if the removable disk 'diskName' is known to have media present.

```
int diskReadSectors(const char *name, unsigned sect, unsigned
count, void *buf)
```

Read 'count' sectors from disk 'name', starting at (zero-based) logical sector number 'sect'. Put the data in memory area 'buf'. This function requires supervisor privilege.

```
int diskWriteSectors(const char *name, unsigned sect, unsigned
count, void *buf)
```

Write 'count' sectors to disk 'name', starting at (zero-based) logical sector number 'sect'. Get the data from memory area 'buf'. This function requires supervisor privilege.

Filesystem functions

```
int filesystemFormat(const char *theDisk, const char *type,
const char *label, int longFormat, progress *prog)
```

Format the logical volume 'theDisk', with a string 'type' representing the preferred filesystem type (for example, "fat", "fat16", "fat32, etc). Label it with 'label'. 'longFormat' will do a sector-by-sector format, if supported, and progress can optionally be monitored by passing a non-NULL progress structure pointer 'prog'. It is optional for filesystem drivers to implement this function.

```
int filesystemClobber(const char *theDisk)
```

Clobber all known filesystem types on the logical volume 'theDisk'. It is optional for filesystem drivers to implement this function.

`int filesystemCheck(const char *name, int force, int repair, progress *prog)`

> Check the filesystem on disk 'name'. If 'force' is non-zero, the filesystem will be checked regardless of whether the filesystem driver thinks it needs to be. If 'repair' is non-zero, the filesystem driver will attempt to repair any errors found. If 'repair' is zero, a non-zero return value may indicate that errors were found. If 'repair' is non-zero, a non-zero return value may indicate that errors were found but could not be fixed. Progress can optionally be monitored by passing a non-NULL progress structure pointer 'prog'. It is optional for filesystem drivers to implement this function.

`int filesystemDefragment(const char *name, progress *prog)`

> Defragment the filesystem on disk 'name'. Progress can optionally be monitored by passing a non-NULL progress structure pointer 'prog'. It is optional for filesystem drivers to implement this function.

`int filesystemMount(const char *name, const char *mp)`

> Mount the filesystem on disk 'name', using the mount point specified by the absolute pathname 'mp'. Note that no file or directory called 'mp' should exist, as the mount function will expect to be able to create it.

`int filesystemUnmount(const char *mp)`

> Unmount the filesystem mounted represented by the mount point 'fs'.

`int filesystemGetFree(const char *fs)`

> Returns the amount of free space on the filesystem represented by the mount point 'fs'.

`unsigned filesystemGetBlockSize(const char *fs)`

> Returns the block size (for example, 512 or 1024) of the filesystem represented by the mount point 'fs'.

File functions

Note that in all of the functions of this section, any reference to pathnames means absolute pathnames, from root. E.g. '/files/myfile', not simply 'myfile'. From the kernel's point of view, 'myfile' might be ambiguous.

`int fileFixupPath(const char *orig, char *new)`

Take the absolute pathname in 'orig' and fix it up. This means eliminating extra file separator characters (for example) and resolving links or '.' or '..' components in the pathname.

```
int fileSeparateLast(const char *origPath, char *pathName, char
*fileName)
```

This function will take a combined pathname/filename string and separate the two. The user will pass in the "combined" string along with two pre-allocated char arrays to hold the resulting separated elements.

```
int fileGetDisk(const char *path, disk *d)
```

Given the file name 'path', return the user space structure for the logical disk that the file resides on.

```
int fileCount(const char *path)
```

Get the count of file entries from the directory referenced by 'path'.

```
int fileFirst(const char *path, file *f)
```

Get the first file from the directory referenced by 'path'. Put the information in the file structure 'f'.

```
int fileNext(const char *path, file *f)
```

Get the next file from the directory referenced by 'path'. 'f' should be a file structure previously filled by a call to either fileFirst() or fileNext().

```
int fileFind(const char *name, file *f)
```

Find the file referenced by 'name', and fill the file data structure 'f' with the results if successful.

```
int fileOpen(const char *name, int mode, file *f)
```

Open the file referenced by 'name' using the file open mode 'mode' (defined in). Update the file data structure 'f' if successful.

```
int fileClose(file *f)
```

Close the previously opened file 'f'.

```
int fileRead(file *f, unsigned blocknum, unsigned blocks, unsigned
char *buff)
```

Read data from the previously opened file 'f'. 'f' should have been opened in a read or read/write mode. Read 'blocks' blocks (see the filesystem functions for information about getting the block size of a given filesystem) and put them in buffer 'buff'.

```
int fileWrite(file *f, unsigned blocknum, unsigned blocks,
unsigned char *buff)
```

Write data to the previously opened file 'f'. 'f' should have been opened in a write or read/write mode. Write 'blocks' blocks (see the filesystem functions for information about getting the block size of a given filesystem) from the buffer 'buff'.

```
int fileDelete(const char *name)
```

Delete the file referenced by the pathname 'name'.

```
int fileDeleteRecursive(const char *name)
```

Recursively delete filesystem items, starting with the one referenced by the pathname 'name'.

```
int fileDeleteSecure(const char *name)
```

Securely delete the file referenced by the pathname 'name'. If supported.

```
int fileMakeDir(const char *name)
```

Create a directory to be referenced by the pathname 'name'.

```
int fileRemoveDir(const char *name)
```

Remove the directory referenced by the pathname 'name'.

```
int fileCopy(const char *src, const char *dest)
```

Copy the file referenced by the pathname 'src' to the pathname 'dest'. This will overwrite 'dest' if it already exists.

```
int fileCopyRecursive(const char *src, const char *dest)
```

Recursively copy the file referenced by the pathname 'src' to the pathname 'dest'. If 'src' is a regular file, the result will be the same as using the non-recursive call. However if it is a directory, all contents of the directory and its subdirectories will be copied. This will overwrite any files in the 'dest' tree if they already exist.

```
int fileMove(const char *src, const char *dest)
```

Move (rename) a file referenced by the pathname 'src' to the pathname 'dest'.

```
int fileTimestamp(const char *name)
```

Update the time stamp on the file referenced by the pathname 'name'

```
int fileGetTemp(file *f)
```

Create and open a temporary file in write mode.

```
int fileStreamOpen(const char *name, int mode, fileStream *f)
```

Open the file referenced by the pathname 'name' for streaming operations, using the open mode 'mode' (defined in). Fills the fileStream data structure 'f' with information needed for subsequent filestream operations.

```
int fileStreamSeek(fileStream *f, int offset)
```

Seek the filestream 'f' to the absolute position 'offset'

```
int fileStreamRead(fileStream *f, unsigned bytes, char *buff)
```

Read 'bytes' bytes from the filestream 'f' and put them into 'buff'.

```
int fileStreamReadLine(fileStream *f, unsigned bytes, char *buff)
```

Read a complete line of text from the filestream 'f', and put up to 'bytes' characters into 'buff'

```
int fileStreamWrite(fileStream *f, unsigned bytes, char *buff)
```

Write 'bytes' bytes from the buffer 'buff' to the filestream 'f'.

```
int fileStreamWriteStr(fileStream *f, char *buff)
```

Write the string in 'buff' to the filestream 'f'

```
int fileStreamWriteLine(fileStream *f, char *buff)
```

Write the string in 'buff' to the filestream 'f', and add a newline at the end

```
int fileStreamFlush(fileStream *f)
```

Flush filestream 'f'.

```
int fileStreamClose(fileStream *f)
```

[Flush and] close the filestream 'f'.

Memory functions

`void *memoryGet(unsigned size, const char *desc)`

> Return a pointer to a new block of memory of size 'size' and (optional) physical alignment 'align', adding the (optional) description 'desc'. If no specific alignment is required, use '0'. Memory allocated using this function is automatically cleared (like 'calloc').

`void *memoryGetPhysical(unsigned size, unsigned align, const char *desc)`

> Return a pointer to a new physical block of memory of size 'size' and (optional) physical alignment 'align', adding the (optional) description 'desc'. If no specific alignment is required, use '0'. Memory allocated using this function is NOT automatically cleared. 'Physical' refers to an actual physical memory address, and is not necessarily useful to external programs.

`int memoryRelease(void *p)`

> Release the memory block starting at the address 'p'. Must have been previously allocated using the memoryRequestBlock() function.

`int memoryReleaseAllByProcId(int pid)`

> Release all memory allocated to/by the process referenced by process ID 'pid'. Only privileged functions can release memory owned by other processes.

`int memoryChangeOwner(int opid, int npid, void *addr, void **naddr)`

> Change the ownership of an allocated block of memory beginning at address 'addr'. 'opid' is the process ID of the currently owning process, and 'npid' is the process ID of the intended new owner. 'naddr' is filled with the new address of the memory (since it changes address spaces in the process). Note that only a privileged process can change memory ownership.

`int memoryGetStats(memoryStats *stats, int kernel)`

> Returns the current memory totals and usage values to the current output stream. If non-zero, the flag 'kernel' will return kernel heap statistics instead of overall system statistics.

`int memoryGetBlocks(memoryBlock *blocksArray, unsigned buffSize, int kernel)`

> Returns a copy of the array of used memory blocks in 'blocksArray', up to 'buffSize' bytes. If non-zero, the flag 'kernel' will return kernel heap blocks instead of overall heap allocations.

Multitasker functions

`int multitaskerCreateProcess(const char *name, int privilege, processImage *execImage)`

Create a new process. 'name' will be the new process' name. 'privilege' is the privilege level. 'execImage' is a processImage structure that describes the loaded location of the file, the program's desired virtual address, entry point, size, etc. If the value returned by the call is a positive integer, the call was successful and the value is the new process' process ID. New processes are created and left in a stopped state, so if you want it to run you will need to set it to a running state ('ready', actually) using the function call multitaskerSetProcessState().

`int multitaskerSpawn(void *addr, const char *name, int numargs, void *args[])`

Spawn a thread from the current process. The starting point of the code (for example, a function address) should be specified as 'addr'. 'name' will be the new thread's name. 'numargs' and 'args' will be passed as the "int argc, char *argv[]) parameters of the new thread. If there are no arguments, these should be 0 and NULL, respectively. If the value returned by the call is a positive integer, the call was successful and the value is the new thread's process ID. New threads are created and made runnable, so there is no need to change its state to activate it.

`int multitaskerGetCurrentProcessId(void)`

Returns the process ID of the calling program.

`int multitaskerGetProcess(int pid, process *proc)`

Returns the process structure for the supplied process ID.

`int multitaskerGetProcessByName(const char *name, process *proc)`

Returns the process structure for the supplied process name

`int multitaskerGetProcesses(void *buffer, unsigned buffSize)`

Fills 'buffer' with up to 'buffSize' bytes' worth of process structures, and returns the number of structures copied.

`int multitaskerSetProcessState(int pid, int state)`

Sets the state of the process referenced by process ID 'pid' to the new state 'state'.

`int multitaskerProcessIsAlive(int pid)`

Returns 1 if the process with the id 'pid' still exists and is in a 'runnable' (viable) state. Returns 0 if the process does not exist or is in a 'finished' state.

```
int multitaskerSetProcessPriority(int pid, int priority)
```

Sets the priority of the process referenced by process ID 'pid' to 'priority'.

```
int multitaskerGetProcessPrivilege(int pid)
```

Gets the privilege level of the process referenced by process ID 'pid'.

```
int multitaskerGetCurrentDirectory(char *buff, int buffsz)
```

Returns the absolute pathname of the calling process' current directory. Puts the value in the buffer 'buff' which is of size 'buffsz'.

```
int multitaskerSetCurrentDirectory(const char *buff)
```

Sets the current directory of the calling process to the absolute pathname 'buff'.

```
objectKey multitaskerGetTextInput(void)
```

Get an object key to refer to the current text input stream of the calling process.

```
int multitaskerSetTextInput(int processId, objectKey key)
```

Set the text input stream of the process referenced by process ID 'processId' to a text stream referenced by the object key 'key'.

```
objectKey multitaskerGetTextOutput(void)
```

Get an object key to refer to the current text output stream of the calling process.

```
int multitaskerSetTextOutput(int processId, objectKey key)
```

Set the text output stream of the process referenced by process ID 'processId' to a text stream referenced by the object key 'key'.

```
int multitaskerDuplicateIO(int pid1, int pid2, int clear)
```

Set 'pid2' to use the same input and output streams as 'pid1', and if 'clear' is non-zero, clear any pending input or output.

```
int multitaskerGetProcessorTime(clock_t *clk)
```

Fill the clock_t structure with the amount of processor time consumed by the calling process.

```
void multitaskerYield(void)
```

Yield the remainder of the current processor timeslice back to the multitasker's scheduler. It's nice to do this when you are waiting for some event, so that your process is not 'hungry' (i.e. hogging processor cycles unnecessarily).

`void multitaskerWait(unsigned ticks)`

Yield the remainder of the current processor timeslice back to the multitasker's scheduler, and wait at least 'ticks' timer ticks before running the calling process again. On the PC, one second is approximately 20 system timer ticks.

`int multitaskerBlock(int pid)`

Yield the remainder of the current processor timeslice back to the multitasker's scheduler, and block on the process referenced by process ID 'pid'. This means that the calling process will not run again until process 'pid' has terminated. The return value of this function is the return value of process 'pid'.

`int multitaskerDetach(void)`

This allows a program to 'daemonize', detaching from the IO streams of its parent and, if applicable, the parent stops blocking. Useful for a process that want to operate in the background, or that doesn't want to be killed if its parent is killed.

`int multitaskerKillProcess(int pid, int force)`

Terminate the process referenced by process ID 'pid'. If 'force' is non-zero, the multitasker will attempt to ignore any errors and dismantle the process with extreme prejudice. The 'force' flag also has the necessary side effect of killing any child threads spawned by process 'pid'. (Otherwise, 'pid' is left in a stopped state until its threads have terminated normally).

`int multitaskerKillByName(const char *name, int force)`

Like multitaskerKillProcess, except that it attempts to kill all instances of processes whose names match 'name'

`int multitaskerTerminate(int code)`

Terminate the calling process, returning the exit code 'code'

`int multitaskerSignalSet(int processId, int sig, int on)`

Set the current process' signal handling enabled (on) or disabled for the specified signal number 'sig'

`int multitaskerSignal(int processId, int sig)`

Send the requested signal 'sig' to the process 'processId'

```
int multitaskerSignalRead(int processId)
```

Returns the number code of the next pending signal for the current process, or 0 if no signals are pending.

Loader functions

```
void *loaderLoad(const char *filename, file *theFile)
```

Load a file referenced by the pathname 'filename', and fill the file data structure 'theFile' with the details. The pointer returned points to the resulting file data.

```
objectKey loaderClassify(const char *fileName, void *fileData, int
size, loaderFileClass *class)
```

Given a file by the name 'fileName', the contents 'fileData', of size 'size', get the kernel loader's idea of the file type. If successful, the return value is non-NULL and the loaderFileClass structure 'class' is filled out with the known information.

```
objectKey loaderClassifyFile(const char *fileName, loaderFileClass
*class)
```

Like loaderClassify(), except the first argument 'fileName' is a file name to classify. Returns the kernel loader's idea of the file type. If successful, the return value is non-NULL and the loaderFileClass structure 'class' is filled out with the known information.

```
loaderSymbolTable *loaderGetSymbols(const char *fileName, int
dynamic)
```

Given a binary executable, library, or object file 'fileName' and a flag 'dynamic', return a loaderSymbolTable structure filled out with the loader symbols from the file. If 'dynamic' is non-zero, only symbols used in dynamic linking will be returned (if the file is not a dynamic library or executable, NULL will be returned). If 'dynamic' is zero, return all symbols found in the file.

```
int loaderLoadProgram(const char *command, int privilege)
```

Run 'command' as a process with the privilege level 'privilege'. If successful, the call's return value is the process ID of the new process. The process is left in a stopped state and must be set to a running state explicitly using the multitasker function multitaskerSetProcessState() or the loader function loaderExecProgram().

```
int loaderLoadLibrary(const char *libraryName)
```

This takes the name of a library file 'libraryName' to load and creates a shared library in the kernel. This function is not especially useful to user programs, since normal shared library loading happens automatically as part of the 'loaderLoadProgram' process.

```
int loaderExecProgram(int processId, int block)
```

Execute the process referenced by process ID 'processId'. If 'block' is non-zero, the calling process will block until process 'pid' has terminated, and the return value of the call is the return value of process 'pid'.

```
int loaderLoadAndExec(const char *command, int privilege, int
block)
```

This function is just for convenience, and is an amalgamation of the loader functions loaderLoadProgram() and loaderExecProgram().

Real-time clock functions

```
int rtcReadSeconds(void)
```

Get the current seconds value.

```
int rtcReadMinutes(void)
```

Get the current minutes value.

```
int rtcReadHours(void)
```

Get the current hours value.

```
int rtcDayOfWeek(unsigned day, unsigned month, unsigned year)
```

Get the current day of the week value.

```
int rtcReadDayOfMonth(void)
```

Get the current day of the month value.

```
int rtcReadMonth(void)
```

Get the current month value.

```
int rtcReadYear(void)
```

Get the current year value.

```
unsigned rtcUptimeSeconds(void)
```

Get the number of seconds the system has been running.

```
int rtcDateTime(struct tm *theTime)
```

Get the current data and time as a tm data structure in 'theTime'.

Random number functions

```
unsigned randomUnformatted(void)
```

Get an unformatted random unsigned number. Just any unsigned number.

```
unsigned randomFormatted(unsigned start, unsigned end)
```

Get a random unsigned number between the start value 'start' and the end value 'end', inclusive.

```
unsigned randomSeededUnformatted(unsigned seed)
```

Get an unformatted random unsigned number, using the random seed 'seed' instead of the kernel's default random seed.

```
unsigned randomSeededFormatted(unsigned seed, unsigned start,
unsigned end)
```

Get a random unsigned number between the start value 'start' and the end value 'end', inclusive, using the random seed 'seed' instead of the kernel's default random seed.

Environment functions

```
int environmentGet(const char *var, char *buf, unsigned bufsz)
```

Get the value of the environment variable named 'var', and put it into the buffer 'buf' of size 'bufsz' if successful.

```
int environmentSet(const char *var, const char *val)
```

Set the environment variable 'var' to the value 'val', overwriting any old value that might have been previously set.

```
int environmentUnset(const char *var)
```

Delete the environment variable 'var'.

```
void environmentDump(void)
```

Print a listing of all the currently set environment variables in the calling process' environment space to the current text output stream.

Raw graphics functions

`int graphicsAreEnabled(void)`

Returns 1 if the kernel is operating in graphics mode.

`int graphicGetModes(videoMode *buffer, unsigned size)`

Get up to 'size' bytes worth of videoMode structures in 'buffer' for the supported video modes of the current hardware.

`int graphicGetMode(videoMode *mode)`

Get the current video mode in 'mode'

`int graphicSetMode(videoMode *mode)`

Set the video mode in 'mode'. Generally this will require a reboot in order to take effect.

`int graphicGetScreenWidth(void)`

Returns the width of the graphics screen.

`int graphicGetScreenHeight(void)`

Returns the height of the graphics screen.

`int graphicCalculateAreaBytes(int width, int height)`

Returns the number of bytes required to allocate a graphic buffer of width 'width' and height 'height'. This is a function of the screen resolution, etc.

`int graphicClearScreen(color *background)`

Clear the screen to the background color 'background'.

`int graphicGetColor(const char *colorName, color *getColor)`

Get the system color 'colorName' and place its values in the color structure 'getColor'. Examples of system color names include 'foreground', 'background', and 'desktop'.

`int graphicSetColor(const char *colorName, color *setColor)`

Set the system color 'colorName' to the values in the color structure 'getColor'. Examples of system color names include 'foreground', 'background', and 'desktop'.

```
int graphicDrawPixel(objectKey buffer, color *foreground, drawMode
mode, int xCoord, int yCoord)
```

Draw a single pixel into the graphic buffer 'buffer', using the color 'foreground', the drawing mode 'drawMode' (for example, 'draw_normal' or 'draw_xor'), the X coordinate 'xCoord' and the Y coordinate 'yCoord'. If 'buffer' is NULL, draw directly onto the screen.

```
int graphicDrawLine(objectKey buffer, color *foreground, drawMode
mode, int startX, int startY, int endX, int endY)
```

Draw a line into the graphic buffer 'buffer', using the color 'foreground', the drawing mode 'drawMode' (for example, 'draw_normal' or 'draw_xor'), the starting X coordinate 'startX', the starting Y coordinate 'startY', the ending X coordinate 'endX' and the ending Y coordinate 'endY'. At the time of writing, only horizontal and vertical lines are supported by the linear framebuffer graphic driver. If 'buffer' is NULL, draw directly onto the screen.

```
int graphicDrawRect(objectKey buffer, color *foreground, drawMode
mode, int xCoord, int yCoord, int width, int height, int
thickness, int fill)
```

Draw a rectangle into the graphic buffer 'buffer', using the color 'foreground', the drawing mode 'drawMode' (for example, 'draw_normal' or 'draw_xor'), the starting X coordinate 'xCoord', the starting Y coordinate 'yCoord', the width 'width', the height 'height', the line thickness 'thickness' and the fill value 'fill'. Non-zero fill value means fill the rectangle. If 'buffer' is NULL, draw directly onto the screen.

```
int graphicDrawOval(objectKey buffer, color *foreground, drawMode
mode, int xCoord, int yCoord, int width, int height, int
thickness, int fill)
```

Draw an oval (circle, whatever) into the graphic buffer 'buffer', using the color 'foreground', the drawing mode 'drawMode' (for example, 'draw_normal' or 'draw_xor'), the starting X coordinate 'xCoord', the starting Y coordinate 'yCoord', the width 'width', the height 'height', the line thickness 'thickness' and the fill value 'fill'. Non-zero fill value means fill the oval. If 'buffer' is NULL, draw directly onto the screen. Currently not supported by the linear framebuffer graphic driver.

```
int graphicDrawImage(objectKey buffer, image *drawImage, drawMode
mode, int xCoord, int yCoord, int xOffset, int yOffset, int width,
int height)
```

Draw the image 'drawImage' into the graphic buffer 'buffer', using the drawing mode 'mode' (for example, 'draw_normal' or 'draw_xor'), the starting X coordinate 'xCoord' and the starting Y coordinate 'yCoord'. The 'xOffset' and 'yOffset' parameters specify an offset into the image to start the drawing (0, 0 to draw the whole image). Similarly the 'width' and 'height' parameters allow you to specify a portion of the image (0, 0 to draw the whole image -- minus any X or Y offsets from the previous parameters). So, for example, to draw only the middle pixel of a 3x3 image, you would specify xOffset=1, yOffset=1, width=1, height=1. If 'buffer' is NULL, draw directly onto the screen.

```
int graphicGetImage(objectKey buffer, image *getImage, int xCoord,
int yCoord, int width, int height)
```

Grab a new image 'getImage' from the graphic buffer 'buffer', using the starting X coordinate 'xCoord', the starting Y coordinate 'yCoord', the width 'width' and the height 'height'. If 'buffer' is NULL, grab the image directly from the screen.

```
int graphicDrawText(objectKey buffer, color *foreground, color
*background, objectKey font, const char *text, drawMode mode, int
xCoord, int yCoord)
```

Draw the text string 'text' into the graphic buffer 'buffer', using the colors 'foreground' and 'background', the font 'font', the drawing mode 'drawMode' (for example, 'draw_normal' or 'draw_xor'), the starting X coordinate 'xCoord', the starting Y coordinate 'yCoord'. If 'buffer' is NULL, draw directly onto the screen. If 'font' is NULL, use the default font.

```
int graphicCopyArea(objectKey buffer, int xCoord1, int yCoord1,
int width, int height, int xCoord2, int yCoord2)
```

Within the graphic buffer 'buffer', copy the area bounded by ('xCoord1', 'yCoord1'), width 'width' and height 'height' to the starting X coordinate 'xCoord2' and the starting Y coordinate 'yCoord2'. If 'buffer' is NULL, copy directly to and from the screen.

```
int graphicClearArea(objectKey buffer, color *background, int
xCoord, int yCoord, int width, int height)
```

Clear the area of the graphic buffer 'buffer' using the background color 'background', using the starting X coordinate 'xCoord', the starting Y coordinate 'yCoord', the width 'width' and the height 'height'. If 'buffer' is NULL, clear the area directly on the screen.

```
int graphicRenderBuffer(objectKey buffer, int drawX, int drawY,
int clipX, int clipY, int clipWidth, int clipHeight)
```

Draw the clip of the buffer 'buffer' onto the screen. Draw it on the screen at starting X coordinate 'drawX' and starting Y coordinate 'drawY'. The buffer clip is bounded by the starting X coordinate 'clipX', the starting Y coordinate 'clipY', the width 'clipWidth' and the height 'clipHeight'. It is not legal for 'buffer' to be NULL in this case.

Windowing system functions

```
int windowLogin(const char *userName)
```

Log the user into the window environment with 'userName'. The return value is the PID of the window shell for this session. The calling program must have supervisor privilege in order to use this function.

```
int windowLogout(void)
```

Log the current user out of the windowing system. This kills the window shell process returned by windowLogin() call.

```
objectKey windowNew(int processId, const char *title)
```

Create a new window, owned by the process 'processId', and with the title 'title'. Returns an object key to reference the window, needed by most other window functions below (such as adding components to the window)

```
objectKey windowNewDialog(objectKey parent, const char *title)
```

Create a dialog window to associate with the parent window 'parent', using the supplied title. In the current implementation, dialog windows are modal, which is the main characteristic distinguishing them from regular windows.

```
int windowDestroy(objectKey window)
```

Destroy the window referenced by the object key 'wndow'

```
int windowUpdateBuffer(void *buffer, int xCoord, int yCoord, int
width, int height)
```

Tells the windowing system to redraw the visible portions of the graphic buffer 'buffer', using the given clip coordinates/size.

```
int windowSetTitle(objectKey window, const char *title)
```

Set the new title of window 'window' to be 'title'.

```
int windowGetSize(objectKey window, int *width, int *height)
```

Get the size of the window 'window', and put the results in 'width' and 'height'.

```
int windowSetSize(objectKey window, int width, int height)
```

Resize the window 'window' to the width 'width' and the height 'height'.

```
int windowGetLocation(objectKey window, int *xCoord, int *yCoord)
```

Get the current screen location of the window 'window' and put it into the coordinate `variables 'xCoord' and 'yCoord'.`

`int windowSetLocation(objectKey window, int xCoord, int yCoord)`

Set the screen location of the window 'window' using the coordinate variables 'xCoord' and 'yCoord'.

`int windowCenter(objectKey window)`

Center 'window' on the screen.

`int windowSnapIcons(objectKey parent)`

If 'container' (either a window or a windowContainer) has icon components inside it, this will snap them to a grid.

`int windowSetHasBorder(objectKey window, int trueFalse)`

Tells the windowing system whether to draw a border around the window 'window'. 'trueFalse' being non-zero means draw a border. Windows have borders by default.

`int windowSetHasTitleBar(objectKey window, int trueFalse)`

Tells the windowing system whether to draw a title bar on the window 'window'. 'trueFalse' being non-zero means draw a title bar. Windows have title bars by default.

`int windowSetMovable(objectKey window, int trueFalse)`

Tells the windowing system whether the window 'window' should be movable by the user (i.e. clicking and dragging it). 'trueFalse' being non-zero means the window is movable. Windows are movable by default.

`int windowSetResizable(objectKey window, int trueFalse)`

Tells the windowing system whether to allow 'window' to be resized by the user.

`int windowSetHasMinimizeButton(objectKey window,`

Tells the windowing system whether to draw a minimize button on the title bar of the window 'window'. 'trueFalse' being non-zero means draw a minimize button. Windows have minimize buttons by default, as long as they have a title bar. If there is no title bar, then this function has no effect.

`int windowSetHasCloseButton(objectKey window, int trueFalse)`

Tells the windowing system whether to draw a close button on the title bar of the window 'window'. 'trueFalse' being non-zero means draw a close button. Windows have close buttons by default, as long as they have a title bar. If there is no title bar, then this function has no effect.

`int windowSetColors(objectKey window, color *background)`

Set the background color of 'window'. If 'color' is NULL, use the default.

`int windowSetVisible(objectKey window, int visible)`

Tell the windowing system whether to make 'window' visible or not. Non-zero 'visible' means make the window visible. When windows are created, they are not visible by default so you can add components, do layout, set the size, etc.

`void windowSetMinimized(objectKey window, int minimized)`

Tell the windowing system whether to make 'window' minimized or not. Non-zero 'minimized' means make the window non-visible, but accessible via the task bar. Zero 'minimized' means restore a minimized window to its normal, visible size.

```
int windowAddConsoleTextArea(objectKey window, componentParameters
*params)
```

Add a console text area component to 'window' using the supplied componentParameters. The console text area is where most kernel logging and error messages are sent, particularly at boot time. Note that there is only one instance of the console text area, and thus it can only exist in one window at a time. Destroying the window is one way to free the console area to be used in another window.

```
void windowRedrawArea(int xCoord, int yCoord, int width, int
height)
```

Tells the windowing system to redraw whatever is supposed to be in the screen area bounded by the supplied coordinates. This might be useful if you were drawing non-window-based things (i.e., things without their own independent graphics buffer) directly onto the screen and you wanted to restore an area to its original contents. For example, the mouse driver uses this method to erase the pointer from its previous position.

`void windowProcessEvent(objectKey event)`

Creates a window event using the supplied event structure. This function is most often used within the kernel, particularly in the mouse and keyboard functions, to signify clicks or key presses. It can, however, be used by external programs to create 'artificial' events. The windowEvent structure specifies the target component and event type.

`int windowComponentEventGet(objectKey key, windowEvent *event)`

Gets a pending window event, if any, applicable to component 'key', and puts the data into the windowEvent structure 'event'. If an event was received, the function returns a positive, non-zero value (the actual value reflects the amount of raw data read from the component's event stream -- not particularly useful to an application). If the return value is zero, no event was pending.

`int windowTileBackground(const char *theFile)`

Load the image file specified by the pathname 'theFile', and if successful, tile the image on the background root window.

`int windowCenterBackground(const char *theFile)`

Load the image file specified by the pathname 'theFile', and if successful, center the image on the background root window.

`int windowScreenShot(image *saveImage)`

Get an image representation of the entire screen in the image data structure 'saveImage'. Note that it is not necessary to allocate memory for the data pointer of the image structure beforehand, as this is done automatically. You should, however, deallocate the data field of the image structure when you are finished with it.

`int windowSaveScreenShot(const char *filename)`

Save a screenshot of the entire screen to the file specified by the pathname 'filename'.

`int windowSetTextOutput(objectKey key)`

Set the text output (and input) of the calling process to the object key of some window component, such as a TextArea or TextField component. You'll need to use this if you want to output text to one of these components or receive input from one.

`int windowComponentSetVisible(objectKey component, int visible)`

Set 'component' visible or non-visible.

`int windowComponentSetEnabled(objectKey component, int enabled)`

Set 'component' enabled or non-enabled; non-enabled components appear greyed-out.

`int windowComponentGetWidth(objectKey component)`

Get the pixel width of the window component 'component'.

`int windowComponentSetWidth(objectKey component, int width)`

Set the pixel width of the window component 'component'

```
int windowComponentGetHeight(objectKey component)
```

Get the pixel height of the window component 'component'.

```
int windowComponentSetHeight(objectKey component, int height)
```

Set the pixel height of the window component 'component'.

```
int windowComponentFocus(objectKey component)
```

Give window component 'component' the focus of its window.

```
int windowComponentDraw(objectKey component)
```

Calls the window component 'component' to redraw itself.

```
int windowComponentGetData(objectKey component, void *buffer, int
size)
```

This is a generic call to get data from the window component 'component', up to 'size' bytes, in the buffer 'buffer'. The size and type of data that a given component will return is totally dependent upon the type and implementation of the component.

```
int windowComponentSetData(objectKey component, void *buffer, int
size)
```

This is a generic call to set data in the window component 'component', up to 'size' bytes, in the buffer 'buffer'. The size and type of data that a given component will use or accept is totally dependent upon the type and implementation of the component.

```
int windowComponentGetSelected(objectKey component, int
*selection)
```

This is a call to get the 'selected' value of the window component 'component'. The type of value returned depends upon the type of component; a list component, for example, will return the 0-based number(s) of its selected item(s). On the other hand, a boolean component such as a checkbox will return 1 if it is currently selected.

```
int windowComponentSetSelected(objectKey component, int selected)
```

This is a call to set the 'selected' value of the window component 'component'. The type of value accepted depends upon the type of component; a list component, for example, will use the 0-based number to select one of its items. On the other hand, a boolean component such as a checkbox will clear itself if 'selected' is 0, and set itself otherwise.

```
objectKey windowNewButton(objectKey parent, const char *label,
image *buttonImage, componentParameters *params)
```

Get a new button component to be placed inside the parent object 'parent', with the given component parameters, and with the (optional) label 'label', or the (optional) image 'buttonImage'. Either 'label' or 'buttonImage' can be used, but not both.

```
objectKey windowNewCanvas(objectKey parent, int width, int height,
componentParameters *params)
```

Get a new canvas component, to be placed inside the parent object 'parent', using the supplied width and height, with the given component parameters. Canvas components are areas which will allow drawing operations, for example to show line drawings or unique graphical elements.

```
objectKey windowNewCheckbox(objectKey parent, const char *text,
componentParameters *params)
```

Get a new checkbox component, to be placed inside the parent object 'parent', using the accompanying text 'text', and with the given component parameters.

```
objectKey windowNewContainer(objectKey parent, const char *name,
componentParameters *params)
```

Get a new container component, to be placed inside the parent object 'parent', using the name 'name', and with the given component parameters. Containers are useful for layout when a simple grid is not sufficient. Each container has its own internal grid layout (for components it contains) and external grid parameters for placing it inside a window or another container. Containers can be nested arbitrarily. This allows limitless control over a complex window layout.

```
objectKey windowNewIcon(objectKey parent, image *iconImage, const
char *label, componentParameters *params)
```

Get a new icon component to be placed inside the parent object 'parent', using the image data structure 'iconImage' and the label 'label', and with the given component parameters 'params'.

```
objectKey windowNewImage(objectKey parent, image *baseImage,
drawMode mode, componentParameters *params)
```

Get a new image component to be placed inside the parent object 'parent', using the image data structure 'baseImage', and with the given component parameters 'params'.

```
objectKey windowNewList(objectKey parent, windowListType type, int
rows, int columns, int multiple, listItemParameters *items, int
numItems, componentParameters *params)
```

Get a new window list component to be placed inside the parent object 'parent', using the component parameters 'params'. 'type' specifies the type of list (see for possibilities), 'rows' and 'columns' specify the size of the list and layout of the list items, 'multiple' allows multiple selections if non-zero, and 'items' is an array of 'numItems' list item parameters.

```
objectKey windowNewListItem(objectKey parent, listItemParameters
*item, componentParameters *params)
```

Get a new list item component to be placed inside the parent object 'parent', using the list item parameters 'item', and the component parameters 'params'.

```
objectKey windowNewMenu(objectKey parent, const char *name,
componentParameters *params)
```

Get a new menu component to be placed inside the parent object 'parent', using the name 'name' (which will be the header of the menu) and the component parameters 'params', and with the given component parameters 'params'. A menu component is an instance of a container, typically contains menu item components, and is typically added to a menu bar component.

```
objectKey windowNewMenuBar(objectKey parent, componentParameters
*params)
```

Get a new menu bar component to be placed inside the parent object 'parent', using the component parameters 'params'. A menu bar component is an instance of a container, and typically contains menu components.

```
objectKey windowNewMenuItem(objectKey parent, const char *text,
componentParameters *params)
```

Get a new menu item component to be placed inside the parent object 'parent', using the string 'text' and the component parameters 'params'. A menu item component is typically added to menu components, which are in turn added to menu bar components.

```
objectKey windowNewPasswordField(objectKey parent, int columns,
componentParameters *params)
```

Get a new password field component to be placed inside the parent object 'parent', using 'columns' columns and the component parameters 'params'. A password field component is a special case of a text field component, and behaves the same way except that typed characters are shown as asterisks (*).

```
objectKey windowNewProgressBar(objectKey parent,
componentParameters *params)
```

Get a new progress bar component to be placed inside the parent object 'parent', using the component parameters 'params'. Use the windowComponentSetData() function to set the percentage of progress.

```
objectKey windowNewRadioButton(objectKey parent, int rows, int
columns, char *items[], int numItems, componentParameters *params)
```

Get a new radio button component to be placed inside the parent object 'parent', using the component parameters 'params'. 'rows' and 'columns' specify the size and layout of the items, and 'numItems' specifies the number of strings in the array 'items', which specifies the different radio button choices. The windowComponentSetSelected() and windowComponentGetSelected() functions can be used to get and set the selected item (numbered from zero, in the order they were supplied in 'items').

```
objectKey windowNewScrollBar(objectKey parent, scrollBarType type,
int width, int height, componentParameters *params)
```

Get a new scroll bar component to be placed inside the parent object 'parent', with the scroll bar type 'type', and the given component parameters 'params'.

```
objectKey windowNewTextArea(objectKey parent, int columns, int
rows, int bufferLines, componentParameters *params)
```

Get a new text area component to be placed inside the parent object 'parent', with the given component parameters 'params'. The 'columns' and 'rows' are the visible portion, and 'bufferLines' is the number of extra lines of scrollback memory. If 'font' is NULL, the default font will be used.

```
objectKey windowNewTextField(objectKey parent, int columns,
componentParameters *params)
```

Get a new text field component to be placed inside the parent object 'parent', using the number of columns 'columns' and with the given component parameters 'params'. Text field components are essentially 1-line 'text area' components. If the params 'font' is NULL, the default font will be used.

```
objectKey windowNewTextLabel(objectKey parent, const char *text,
componentParameters *params)
```

Get a new text labelComponent to be placed inside the parent object 'parent', with the given component parameters 'params', and using the text string 'text'. If the params 'font' is NULL, the default font will be used.

```
void windowDebugLayout(objectKey window)
```

This function draws grid boxes around all the grid cells containing components (or parts thereof)

User functions

`int userAuthenticate(const char *name, const char *password)`

Given the user 'name', return 0 if 'password' is the correct password.

`int userLogin(const char *name, const char *password)`

Log the user 'name' into the system, using the password 'password'. Calling this function requires supervisor privilege level.

`int userLogout(const char *name)`

Log the user 'name' out of the system. This can only be called by a process with supervisor privilege, or one running as the same user being logged out.

`int userGetNames(char *buffer, unsigned bufferSize)`

Fill the buffer 'buffer' with the names of all users, up to 'bufferSize' bytes.

`int userAdd(const char *name, const char *password)`

Add the user 'name' with the password 'password'

`int userDelete(const char *name)`

Delete the user 'name'

```
int userSetPassword(const char *name, const char *oldPass, const
char *newPass)
```

Set the password of user 'name'. If the calling program is not supervisor privilege, the correct old password must be supplied in 'oldPass'. The new password is supplied in 'newPass'.

`int userGetPrivilege(const char *name)`

Get the privilege level of the user represented by 'name'.

`int userGetPid(void)`

Get the process ID of the current user's 'login process'.

`int userSetPid(const char *name, int pid)`

Set the login PID of user 'name' to 'pid'. This is the process that gets killed when the user indicates that they want to logout. In graphical mode this will typically be the PID of the window shell pid, and in text mode it will be the PID of the login VSH shell.

`int userFileAdd(const char *passFile, const char *userName, const char *password)`

Add a user to the designated password file, with the given name and password. This can only be done by a privileged user.

`int userFileDelete(const char *passFile, const char *userName)`

Remove a user from the designated password file. This can only be done by a privileged user

`int userFileSetPassword(const char *passFile, const char *userName, const char *oldPass, const char *newPass)`

Set the password of user 'userName' in the designated password file. If the calling program is not supervisor privilege, the correct old password must be supplied in 'oldPass'. The new password is supplied in 'newPass'.

Network functions

`int networkDeviceGetCount(void)`

Returns the count of network devices

`int networkDeviceGet(const char *name, networkDevice *dev)`

Returns the user-space portion of the requested (by 'name') network device in 'dev'.

`int networkInitialized(void)`

Returns 1 if networking is currently enabled.

`int networkInitialize(void)`

Initialize and start networking.

`int networkShutdown(void)`

Shut down networking.

`objectKey networkOpen(int mode, networkAddress *address, networkFilter *filter)`

Opens a connection for network communication. The 'type' and 'mode' arguments describe the kind of connection to make (see possiblilities in the file . If applicable, 'address' specifies the network address of the remote host to connect to. If applicable, the 'localPort' and 'remotePort' arguments specify the TCP/UDP ports to use.

`int networkClose(objectKey connection)`

Close the specified, previously-opened network connection.

`int networkCount(objectKey connection)`

Given a network connection, return the number of bytes currently pending in the input stream

`int networkRead(objectKey connection, unsigned char *buffer, unsigned bufferSize)`

Given a network connection, a buffer, and a buffer size, read up to 'bufferSize' bytes (or the number of bytes available in the connection's input stream) and return the number read. The connection must be initiated using the networkConnectionOpen() function.

`int networkWrite(objectKey connection, unsigned char *buffer, unsigned bufferSize)`

Given a network connection, a buffer, and a buffer size, write up to 'bufferSize' bytes from 'buffer' to the connection's output. The connection must be initiated using the networkConnectionOpen() function.

`int networkPing(objectKey connection, int seqNum, unsigned char *buffer, unsigned bufferSize)`

Send an ICMP "echo request" packet to the host at the network address 'destAddress', with the (optional) sequence number 'seqNum'. The 'buffer' and 'bufferSize' arguments point to the location of data to send in the ping packet. The content of the data is mostly irrelevant, except that it can be checked to ensure the same data is returned by the ping reply from the remote host.

Miscellaneous functions

`int fontGetDefault(objectKey *pointer)`

Get an object key in 'pointer' to refer to the current default font.

`int fontSetDefault(const char *name)`

Set the default font for the system to the font with the name 'name'. The font must previously have been loaded by the system, for example using the fontLoad() function.

`int fontLoad(const char* filename, const char *fontname, objectKey *pointer, int fixedWidth)`

Load the font from the font file 'filename', give it the font name 'fontname' for future reference, and return an object key for the font in 'pointer' if successful. The integer 'fixedWidth' argument should be non-zero if you want each character of the font to have uniform width (i.e. an 'i' character will be padded with empty space so that it takes up the same width as, for example, a 'W' character).

```
int fontGetPrintedWidth(objectKey font, const char *string)
```

Given the supplied string, return the screen width that the text will consume given the font 'font'. Useful for placing text when using a variable-width font, but not very useful otherwise.

```
int imageLoad(const char *filename, int width, int height, image
*loadImage)
```

Try to load the bitmap image file 'filename' (with the specified 'width' and 'height' if possible -- zeros indicate no preference), and if successful, save the data in the image data structure 'loadImage'.

```
int imageSave(const char *filename, int format, image *saveImage)
```

Save the image data structure 'saveImage' using the image format 'format' to the file 'fileName'. Image format codes are found in the file

```
int shutdown(int reboot, int nice)
```

Shut down the system. If 'reboot' is non-zero, the system will reboot. If 'nice' is zero, the shutdown will be orderly and will abort if serious errors are detected. If 'nice' is non-zero, the system will go down like a kamikaze regardless of errors.

```
const char *version(void)
```

Get the kernel's version string.

```
int encryptMD5(const char *in, char *out)
```

Given the input string 'in', return the encrypted numerical message digest in the buffer 'out'.

```
int lockGet(lock *getLock)
```

Get an exclusive lock based on the lock structure 'getLock'.

```
int lockRelease(lock *relLock)
```

Release a lock on the lock structure 'lock' previously obtained with a call to the lockGet() function.

`int lockVerify(lock *verLock)`

Verify that a lock on the lock structure 'verLock' is still valid. This can be useful for retrying a lock attempt if a previous one failed; if the process that was previously holding the lock has failed, this will release the lock.

`int variableListCreate(variableList *list)`

Set up a new variable list structure.

`int variableListDestroy(variableList *list)`

Deallocate a variable list structure previously allocated by a call to variableListCreate() or configurationReader()

`int variableListGet(variableList *list, const char *var, char *buffer, unsigned buffSize)`

Get the value of the variable 'var' from the variable list 'list' in the buffer 'buffer', up to 'buffSize' bytes.

`int variableListSet(variableList *list, const char *var, const char *value)`

Set the value of the variable 'var' to the value 'value'.

`int variableListUnset(variableList *list, const char *var)`

Remove the variable 'var' from the variable list 'list'.

`int configurationReader(const char *fileName, variableList *list)`

Read the contents of the configuration file 'fileName', and return the data in the variable list structure 'list'. Configuration files are simple properties files, consisting of lines of the format "variable=value"

`int configurationWriter(const char *fileName, variableList *list)`

Write the contents of the variable list 'list' to the configuration file 'fileName'. Configuration files are simple properties files, consisting of lines of the format "variable=value". If the configuration file already exists, the configuration writer will attempt to preserve comment lines (beginning with '#') and formatting whitespace.

`int keyboardGetMaps(char *buffer, unsigned size)`

Get a listing of the names of all available keyboard mappings. The buffer is filled up to 'size' bytes with descriptive names, such as "English (UK)". Each string is NULL-terminated, and the return value of the call is the number of strings copied. The first string returned is the current map.

```
int keyboardSetMap(const char *name)
```

Set the keyboard mapping to the supplied 'name'. The normal procedure would be to first call the keyboardGetMaps() function, get the list of supported mappings, and then call this function with one of those names. Only a name returned by the keyboardGetMaps function is valid in this scenario.

```
int deviceTreeGetCount(void)
```

Returns the number of devices in the kernel's device tree.

```
int deviceTreeGetRoot(device *rootDev)
```

Returns the user-space portion of the device tree root device in the structure 'rootDev'.

```
int deviceTreeGetChild(device *parentDev, device *childDev)
```

Returns the user-space portion of the first child device of 'parentDev' in the structure 'childDev'.

```
int deviceTreeGetNext(device *siblingDev)
```

Returns the user-space portion of the next sibling device of the supplied device 'siblingDev' in the same data structure.

```
int mouseLoadPointer(const char *pointerName, const char
*fileName)
```

Tells the mouse driver code to load the mouse pointer 'pointerName' from the file 'fileName'.

```
int mouseSwitchPointer(const char *pointerName)
```

Tells the mouse driver code to switch to the mouse pointer with the given name (as specified by the pointer name argument to mouseLoadPointer).

THE VISOPSYS WINDOW LIBRARY (Version 0.6)

The window library is a set of functions to aid GUI development on the Visopsys platform. At present the list of functions is small, but it does provide very useful functionality. This includes an interface for registering window event callbacks for GUI components, and a 'run' function to poll for such events.

The functions are defined in the header file <sys/window.h> and the code is contained in libwindow.a (link with '-lwindow').

```
objectKey windowNewBannerDialog(objectKey parentWindow, const char
*title, const char *message)
```

Create a 'banner' dialog box, with the parent window 'parentWindow', and the given titlebar text and main message. This is the very simplest kind of dialog; it just contains the supplied message with no acknowledgement mechanism for the user. If 'parentWindow' is NULL, the dialog box is actually created as an independent window that looks the same as a dialog. This is a non-blocking call that returns the object key of the dialog window. The caller must destroy the window when finished with it.

```
void windowCenterDialog(objectKey parentWindow, objectKey
dialogWindow)
```

Center a dialog window. The first object key is the parent window, and the second is the dialog window. This function can be used to center a regular window on the screen if the first objectKey argument is NULL.

```
int windowNewChoiceDialog(objectKey parentWindow, const char
*title, const char *message, char *choiceStrings[], int
numChoices, int defaultChoice)
```

Create a 'choice' dialog box, with the parent window 'parentWindow', the given titlebar text and main message, and 'numChoices' choices, as specified by the 'choiceStrings'. 'default' is the default focussed selection. The dialog will have a button for each choice. If the user chooses one of the choices, the function returns the 0-based index of the choice. Otherwise it returns negative. If 'parentWindow' is NULL, the dialog box is actually created as an independent window that looks the same as a dialog. This is a blocking call that returns when the user closes the dialog window (i.e. the dialog is 'modal').

```
int windowNewColorDialog(objectKey parentWindow, color
*pickedColor)
```

Create an 'color chooser' dialog box, with the parent window 'parentWindow', and a pointer to the color structure 'pickedColor'. Currently the window consists of red/green/blue sliders and a canvas displaying the current color. The initial color displayed will be whatever is supplied in 'pickedColor'. If 'parentWindow' is NULL, the dialog box is actually created as an independent window that looks the same as a dialog. This is a blocking call that returns when the user closes the dialog window (i.e. the dialog is 'modal').

```
int windowNewFileDialog(objectKey parentWindow, const char *title,
const char *message, const char *startDir, char *fileName,
unsigned maxLength)
```

Create a 'file' dialog box, with the parent window 'parentWindow', and the given titlebar text and main message. If 'startDir' is a non-NULL directory name, the dialog will initially display the contents of that directory. The dialog will have a file selection area, an 'OK' button and a 'CANCEL' button. If the user presses OK or ENTER, the function returns the value 1 and copies the filename into the fileName buffer. Otherwise it returns 0 and puts a NULL string into fileName. If 'parentWindow' is NULL, the dialog box is actually created as an independent window that looks the same as a dialog. This is a blocking call that returns when the user closes the dialog window (i.e. the dialog is 'modal').

```
objectKey windowNewFileList(objectKey parent, windowListType type,
int rows, int columns, const char *directory, int flags, void
*callback, componentParameters *params)
```

Create a new file list widget with the parent window 'parent', the window list type 'type' (windowlist_textonly or windowlist_icononly is currently supported), of height 'rows' and width 'columns', the name of the starting location 'directory', flags (such as WINFILEBROWSE_CAN_CD or WINFILEBROWSE_CAN_DEL -- see sys/window.h), a function 'callback' for when the status changes, and component parameters 'params'.

```
int windowUpdateFileList(objectKey fileList, const char
*directory)
```

Update the supplied file list 'fileList', with the location 'directory'. This is useful for changing the current directory, for example.

```
int windowDestroyFileList(objectKey fileList)
```

Clear the event handler for the file list widget 'fileList', and destroy and deallocate the widget.

```
int windowClearEventHandlers(void)
```

Remove all the callback event handlers registered with the windowRegisterEventHandler() function.

```
int windowRegisterEventHandler(objectKey key, void
(*function)(objectKey, windowEvent *))
```

Register a callback function as an event handler for the GUI object 'key'. The GUI object can be a window component, or a window for example. GUI components are typically the target of mouse click or key press events, whereas windows typically receive 'close window' events. For example, if you create a button component in a window, you should call windowRegisterEventHandler() to receive a callback when the button is pushed by a user. You can use the same callback function for all your objects if you wish -- the objectKey of the target component can always be found in the windowEvent passed to your callback function. It is necessary to use one of the 'run' functions, below, such as windowGuiRun() or windowGuiThread() in order to receive the callbacks.

`int windowClearEventHandler(objectKey key)`

Remove a callback event handler registered with the windowRegisterEventHandler() function.

`void windowGuiRun(void)`

Run the GUI windowEvent polling as a blocking call. In other words, use this function when your program has completed its setup code, and simply needs to watch for GUI events such as mouse clicks, key presses, and window closures. If your program needs to do other processing (independently of windowEvents) you should use the windowGuiThread() function instead.

`int windowGuiThread(void)`

Run the GUI windowEvent polling as a non-blocking call. In other words, this function will launch a separate thread to monitor for GUI events and return control to your program. Your program can then continue execution -- independent of GUI windowEvents. If your program doesn't need to do any processing after setting up all its window components and event callbacks, use the windowGuiRun() function instead.

`int windowGuiThreadPid(void)`

Returns the current GUI thread PID, if applicable, or else 0.

`void windowGuiStop(void)`

Stop GUI event polling which has been started by a previous call to one of the 'run' functions, such as windowGuiRun() or windowGuiThread().

```
int windowNewInfoDialog(objectKey parentWindow, const char *title,
const char *message)
```

Create an 'info' dialog box, with the parent window 'parentWindow', and the given titlebar text and main message. The dialog will have a single 'OK' button for the user to acknowledge. If 'parentWindow' is NULL, the dialog box is actually created as an independent window that looks the same as a dialog. This is a blocking call that returns when the user closes the dialog window (i.e. the dialog is 'modal').

```
int windowNewErrorDialog(objectKey parentWindow, const char
*title, const char *message)
```

Create an 'error' dialog box, with the parent window 'parentWindow', and the given titlebar text and main message. The dialog will have a single 'OK' button for the user to acknowledge. If 'parentWindow' is NULL, the dialog box is actually created as an independent window that looks the same as a dialog. This is a blocking call that returns when the user closes the dialog window (i.e. the dialog is 'modal').

```
objectKey windowNewProgressDialog(objectKey parentWindow, const
char *title, progress *tmpProg)
```

Create a 'progress' dialog box, with the parent window 'parentWindow', and the given titlebar text and progress structure. The dialog creates a thread which monitors the progress structure for changes, and updates the progress bar and status message appropriately. If the operation is interruptible, it will show a 'CANCEL' button. If 'parentWindow' is NULL, the dialog box is actually created as an independent window that looks the same as a dialog. This is a non-blocking call that returns immediately (but the dialog box itself is 'modal'). A call to this function should eventually be followed by a call to windowProgressDialogDestroy() in order to destroy and deallocate the window.

```
int windowProgressDialogDestroy(objectKey window)
```

Given the objectKey for a progress dialog 'window' previously returned by windowNewProgressDialog(), destroy and deallocate the window.

```
int windowNewPromptDialog(objectKey parentWindow, const char
*title, const char *message, int rows, int columns, char *buffer)
```

Create a 'prompt' dialog box, with the parent window 'parentWindow', and the given titlebar text and main message. The dialog will have a single text field for the user to enter data. If 'parentWindow' is NULL, the dialog box is actually created as an independent window that looks the same as a dialog. This is a blocking call that returns when the user closes the dialog window (i.e. the dialog is 'modal').

```
int windowNewPasswordDialog(objectKey parentWindow, const char
*title, const char *message, int columns, char *buffer)
```

Create a 'password' dialog box, with the parent window 'parentWindow', and the given titlebar text and main message. The dialog will have a single password field. If 'parentWindow' is NULL, the dialog box is actually created as an independent window that looks the same as a dialog. This is a blocking call that returns when the user closes the dialog window (i.e. the dialog is 'modal').

```
int windowNewQueryDialog(objectKey parentWindow, const char
*title, const char *message)
```

Create a 'query' dialog box, with the parent window 'parentWindow', and the given titlebar text and main message. The dialog will have an 'OK' button and a 'CANCEL' button. If the user presses OK, the function returns the value 1. Otherwise it returns 0. If 'parentWindow' is NULL, the dialog box is actually created as an independent window that looks the same as a dialog. This is a blocking call that returns when the user closes the dialog window (i.e. the dialog is 'modal').

THE VISOPSYS SHELL LIBRARY (Version 0.6)

The shell library is a small set of functions created for the Visopsys shell, /programs/vsh, and provided as a library for other programs to use.

The functions are defined in the header file <sys/vsh.h> and the code is contained in libvsh.a (link with '-lvsh'). This code also requires a C library to link correctly (link with '-lc').

`void vshCompleteFilename(char *buffer)`

> Attempts to complete a portion of a filename, 'buffer'. The function will append either the remainder of the complete filename, or if possible, some portion thereof. The result simply depends on whether a good completion or partial completion exists. 'buffer' must of course be large enough to contain any potential filename completion.

`int vshCopyFile(const char *srcFile, const char *destFile)`

> Copy the file specified by the name 'srcFile' to the filename 'destFile'. Both filenames must be absolute pathnames, beginning with '/'.

`int vshCursorMenu(const char *prompt, int numItems, char *items[], int defaultSelection)`

> This will create a pretty cursor-changeable text menu with the supplied 'prompt' string at the stop. Returns the integer (zero-based) selected item number, or else negative on error or no selection.

`int vshDeleteFile(const char *deleteFile)`

> Delete the file specified by the name 'deleteFile'. 'deleteFile' must be an absolute pathname, beginning with '/'.

`int vshDumpFile(const char *fileName)`

> Print the contents of the file, specified by 'fileName', to standard output. 'fileName' must be an absolute pathname, beginning with '/'.

`int vshFileList(const char *itemName)`

> Print a listing of a file or directory named 'itemName'. 'itemName' must be an absolute pathname, beginning with '/'.

`void vshMakeAbsolutePath(const char *orig, char *new)`

> Turns a filename, specified by 'orig', into an absolute pathname 'new'. This basically just amounts to prepending the name of the current directory (plus a '/') to the supplied name. 'new' must be a buffer large enough to hold the entire filename.

`int vshMoveFile(const char *srcFile, const char *destFile)`

Move (rename) the file specified by the name 'srcFile' to the destination 'destFile'. Both filenames must be absolute pathnames -- beginning with '/' -- and must be within the same filesystem.

`int vshParseCommand(char *commandLine, char *command, int *argCount, char *args[])`

Attempts to take a raw 'commandLine' string and parse it into a command filename and arguments, suitable for passing to the kernel API functionn loaderLoadAndExec. The commandLine string will be modified, with NULLs placed at the end of each argument. 'command' must be a buffer suitable for a full filename. 'argCount' will receive the number of argument pointers placed in the 'args' array. Returns 0 on success, negative otherwise.

`void vshPasswordPrompt(const char *prompt, char *buffer)`

Produces a text-mode prompt for the user to enter a password. The prompt message is the first parameter, and a buffer to contain the result is the second parameter.

`void vshPrintDate(char *buffer, unsigned unformattedDate)`

Print the packed date value, specified by the unsigned integer 'unformattedDate' -- such as that found in the file.modifiedDate field -- into 'buffer' in a (for now, arbitrary) human-readable format.

`void vshPrintTime(char *buffer, unsigned unformattedTime)`

Print the packed time value, specified by the unsigned integer 'unformattedTime' -- such as that found in the file.modifiedTime field -- into 'buffer' in a (for now, arbitrary) human-readable format to standard output.

`int vshProgressBar(progress *tmpProg)`

Given a progress structure 'tmpProg', make a text progress bar that monitors the structure and updates itself (in a non-blocking way). After the operation has completed, vshProgressBarDestroy() should be called to shut down the thread.

`int vshProgressBarDestroy(progress *tmpProg)`

Given a progress structure 'tmpProg', indicate 100%, shut down and deallocate anything associated with a previous call to vshProgressBar().

`int vshSearchPath(const char *orig, char *new)`

Search the current path (defined by the PATH environment variable) for the first occurrence of the filename specified in 'orig', and place the complete, absolete pathname result in 'new'. If a match is found, the function returns zero. Otherwise, it returns a negative error code. 'new' must be large enough to hold the complete absolute filename of any match found.