# THE VISOPSYS KERNEL API (Version 0.3)

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.

`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.

`int textClearScreen(void)`

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

`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 diskGetInfo(disk *d)`

Get information about the logical disk volume named by the disk structure's 'name' field and fill in the remainder of the disk structure d.

`int diskGetPhysicalInfo(disk *d)`

Get information about the physical disk device named by the disk structure's 'name' field and fill in the remainder of the disk structure d.

`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 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'.

```
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'.

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

Read 'count' sectors from disk 'name', starting at (zero-based) absolute sector number 'sect'. Put the data in memory area 'buf'. This function requires supervisor privilege and is used to read outside the logical confines of a volume, such as a hard disk partition. Not very useful unless you know what you're doing.

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

Write 'count' sectors to disk 'name', starting at (zero-based) absolute sector number 'sect'. Get the data from memory area 'buf'. This function requires supervisor privilege and is used to write outside the logical confines of a volume, such as a hard disk partition. Don't use this unless you know what you're doing.

**Filesystem functions**

```
int filesystemFormat(const char *disk, const char *type, const
char *label, int longFormat)
```

Format the logical volume 'disk', 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. It is optional for filesystem drivers to implement this function.

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

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. It is optional for filesystem drivers to implement this function.

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

Defragment the filesystem on disk 'name'. 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 filesystemNumberMounted(void)`

Returns the number of filesystems currently mounted.

`void filesystemFirstFilesystem(char *buff)`

Returns the mount point of the first mounted filesystem in 'buff'. Normally this will be the root filesystem "/".

`void filesystemNextFilesystem(char *buff)`

Returns the mount point of the next mounted filesystem as returned by a previous call to either filesystemFirstFilesystem()or filesystemNextFilesystem()

`int filesystemGetFree(const char *fs)`

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

`unsigned int 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 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 <sys/file.h>). Update the file data structure 'f' if successful.

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

Close the previously opened file 'f'.

```
int fileRead(file *f, unsigned int blocknum, unsigned int 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 int blocknum, unsigned int 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 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 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 <sys/file.h>). Fills the fileStream data structure 'f' with information needed for subsequent file stream operations.

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

Seek the file stream 'f' to the absolute position 'offset'

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

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

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

Write 'bytes' bytes from the buffer 'buff' to the file stream 'f'.

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

Flush file stream 'f'.

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

[Flush and] close the file stream 'f'.

**Memory functions**

void memoryPrintUsage(int kernel)

Prints the current memory usage statistics to the current output stream. If non-zero, the flag 'kernel' will show usage of kernel dynamic memory as well.

```
void *memoryGet(unsigned int size, unsigned int align, 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 int size, unsigned int 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.

**Multitasker functions**

`int multitaskerCreateProcess(void *addr, unsigned int size, const char *name, int numargs, void *args)`

> Create a new process. The code should have been loaded at the address 'addr' and be of size 'size'. 'name' will be the new process' name. 'numargs' and 'args' will be passed as the "int argc, char *argv[]) parameters of the new process. 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 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 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 multitaskerGetCurrentProcessId(void)`

> Returns the process ID of the calling program.

`int multitaskerGetProcessOwner(int pid)`

> Returns the user ID of the user that owns the process referenced by process ID 'pid'.

`const char *multitaskerGetProcessName(int pid)`

> Returns the process name of the process referenced by process ID 'pid'.

`int multitaskerGetProcessState(int pid, int *statep)`

> Gets the state of the process referenced by process ID 'pid'. Puts the result in 'statep'.

`int multitaskerSetProcessState(int pid, int state)`

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

`int multitaskerGetProcessPriority(int pid)`

> Gets the priority of the process referenced by process ID 'pid'.

`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(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 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 int 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 wants 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 multitaskerTerminate(int code)`

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

`void multitaskerDumpProcessList(void)`

Print a listing of all current processes to the current text output stream. Might not be the current output stream of the calling process, but rather the console output stream. This could be considered a bug, but is more of a "currently necessary peculiarity".

**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.

```
int loaderLoadProgram(const char *userProgram, int privilege, int
argc, char *argv[])
```

Load the file referenced by the pathname 'userProgram' as a process with the privilege level 'privilege'. Pass the arguments 'argc' and 'argv'. If there are no arguments, these should be 0 and NULL, respectively. 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 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 *name, int privilege, int argc, char *argv[], 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 rtcReadDayOfWeek(void)`

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 int rtcUptimeSeconds(void)
```

Get the number of seconds the system has been running.

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

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

**Random number functions**

```
unsigned int randomUnformatted(void)
```

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

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

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

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

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

```
unsigned int randomSeededFormatted(unsigned int seed, unsigned int
start, unsigned int 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 int 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.

```
unsigned graphicGetScreenWidth(void)
```

Returns the width of the graphics screen.

```
unsigned graphicGetScreenHeight(void)
```

Returns the height of the graphics screen.

```
unsigned graphicCalculateAreaBytes(unsigned width, unsigned
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 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, unsigned width, unsigned height,
unsigned 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, unsigned width, unsigned height,
unsigned 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, int
xCoord, int yCoord, unsigned xOffset, unsigned yOffset, unsigned
width, unsigned height)
```

Draw the image 'drawImage' into the graphic buffer 'buffer', using 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, unsigned width, unsigned 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, objectKey
font, const char *text, drawMode mode, int xCoord, int yCoord)
```

Draw the text string 'text' into the graphic buffer 'buffer', using the color 'foreground', 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,
unsigned width, unsigned 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, unsigned width, unsigned 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, unsigned clipWidth, unsigned 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.

**Window manager functions**

```
int windowManagerStart(void)
```

Starts the window manager. Not useful for most external programa.

```
int windowManagerLogin(const char *userName, const char *passwd)
```

Log the user into the window environment with 'userName' and 'passwd'.

```
int windowManagerLogout(void)
```

Log the current user out of the window manager.

```
objectKey windowManagerNewWindow(int processId, char *title, int
xCoord, int yCoord, int width, int height)
```

Create a new window, owned by the process referenced by the process ID 'processId'. Set the window title to 'title', and place it initially at the specified coordinates with the given width and height. Returns an object key to referenc the window, needed by most other window manager functions below.

```
objectKey windowManagerNewDialog(objectKey parent, char *title,
int xCoord, int yCoord, int width, int height)
```

Create a dialog window to associate with the regular window 'parent', using the supplied title and coordinates.

```
int windowManagerDestroyWindow(objectKey window)
```

Destroy the window referenced by the object key 'wndow'

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

Tells the window manager to redraw the visible portions of the window's graphic buffer 'buffer' and 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, unsigned *width, unsigned
*height)
```

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

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

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

```
int windowAutoSize(objectKey window)
```

Automatically set the size of window 'window' based on the sizes and locations of the window components it contains.

```
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 windowSetHasBorder(objectKey window, int trueFalse)
```

Tells the window manager 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 window manager 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 window manager 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 windowSetHasCloseButton(objectKey window, int trueFalse)
```

Tells the window manager 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 windowLayout(objectKey window)
```

Do the layout of the window 'window' after all the components have been added. You really should call this function before displaying a window, or else all the window components will be squished together in the top corner of the window, ignoring any grid coordinates you have set in the 'componentParameters' structure of an windowAdd*Component() call. If you are going to use windowAutoSize() to size the window, you should do so after this call.

```
int windowSetVisible(objectKey window, int visible)
```

Tell the window manager whether to make the window '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.

```
int windowAddComponent(objectKey window, objectKey component,
componentParameters *params)
```

Add a window component 'component' to the window 'window' using the parameters specified in the componentParameters structure 'params'. You should probably use the windowAddClientComponent() function instead, as this function disregards things like the sizes of window title bars and borders, so your window might look funny unless you know what you're doing.

```
int windowAddClientComponent(objectKey window, objectKey
component, componentParameters *params)
```

Add a window component 'component' to the client area of the window 'window' using the parameters specified in the componentParameters structure 'params'. This function is the normal way to add any component to a window.

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

Add a console text area component to the client area of '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 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.

```
unsigned windowComponentGetWidth(objectKey component)
```

Get the pixel width of the window component 'component'. Useful if you are doing your window layout manually.

```
unsigned windowComponentGetHeight(objectKey component)
```

Get the pixel height of the window component 'component'. Useful if you are doing your window layout manually.

```
void windowManagerRedrawArea(int xCoord, int yCoord, unsigned
width, unsigned height)
```

Tells the window manager 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 on the screen and you wanted them to go away later.

```
void windowManagerProcessEvent(windowEvent *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 windowManagerTileBackground(const char *file)
```

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

```
int windowManagerCenterBackground(const char *file)
```

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

```
int windowManagerScreenShot(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 windowManagerSaveScreenShot(const char *filename)
```

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

```
int windowManagerSetTextOutput(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.

```
objectKey windowNewButton(objectKey window, unsigned width,
unsigned height, const char *label, image *buttonImage)
```

Get a new button component to be placed in the window 'window', using the supplied width and height, and with the (optional) label 'label', or the (optional) image 'buttonImage'. Either 'label' or 'buttonImage' can be used, but not both. After creating any window component you should add it to the window by calling the windowAddClientComponent() function.

```
objectKey windowNewIcon(objectKey window, image *iconImage, const
char *label, const char *command)
```

Get a new icon component to be placed in the window 'window', using the image data structure 'iconImage' and the label 'label'. If you want the icon to execute a command when clicked, you should specify it in 'command'. After creating any window component you should add it to the window by calling the windowAddClientComponent() function.

```
objectKey windowNewImage(objectKey window, image *baseImage)
```

Get a new image component to be placed in the window 'window', using the image data structure 'baseImage' . After creating any window component you should add it to the window by calling the windowAddClientComponent() function.

```
objectKey windowNewTextArea(objectKey window, int columns, int
rows, objectKey font)
```

Get a new text area component to be placed in the window 'window', using the number of columns 'columns', the number of rows 'rows', and the font 'font'. If 'font' is NULL, the default font will be used. After creating any window component you should add it to the window by calling the windowAddClientComponent() function.

```
objectKey windowNewTextField(objectKey window, int columns,
objectKey font)
```

Get a new text field component to be placed in the window 'window', using the number of columns 'columns' and the font 'font'. Text field components are essentially 1-line 'text area' components. If 'font' is NULL, the default font will be used. After creating any window component you should add it to the window by calling the windowAddClientComponent() function.

```
objectKey windowNewTextLabel(objectKey window, objectKey font,
const char *text)
```

Get a new text labelComponent to be placed in the window 'window', using the text string 'text' and the font 'font'. If 'font' is NULL, the default font will be used. After creating any window component you should add it to the window by calling the windowAddClientComponent() function.

```
objectKey windowNewTitleBar(objectKey window, unsigned width,
unsigned height)
```

Get a new title bar component to be placed in the window 'window', using the width 'width' and the height 'height'. This is not necessarily useful to external programs as window title bars are created automatically if applicable, and using this function has the potential to produce surprising results. After creating any window component you should add it to the window by calling the windowAddClientComponent() function.

**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)
```

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.

```
unsigned 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 imageLoadBmp(const char *filename, image *loadImage)
```

Try to load the bitmap image file 'filename', and if successful, save the data in the image data structure 'loadImage'.

```
int imageSaveBmp(const char *filename, image *saveImage)
```

Save the image data structure 'saveImage' as a bitmap, to the file 'fileName'.

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

Log the user 'name' into the system, using the process ID 'pid' as the 'login process' of the user (this process gets killed if the user is forcefully logged out, for example). This function requires supervisor privilege level. The login/logout procedure will likely change in a future release.

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

Log the user 'name' out of the system. This can only be called by a process running as the same user being logged out. The login/logout procedure will likely change in a future release.

```
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 shutdown(int type, int nice)
```

Shutdown or reboot the system, according to the value ('shutdown' or 'reboot') 'type'. 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.

## THE VISOPSYS WINDOW LIBRARY (Version 0.3)

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').

```
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.

```
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.

```
void 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.

```
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(). Note that calling this function clears all callbacks registered with the windowRegisterEventHandler() function, so if you want to resume GUI execution you will need to re-register them.

`int windowNewInfoDialog(objectKey parentWindow, char *title, 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, char *title, 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').

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

Create an '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 0.3)

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 vshPrintTime(unsigned unformattedTime)
```

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

```
void vshPrintDate(unsigned unformattedDate)
```

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

```
int vshFileList(const char *itemName)
```

Print a listing of a file or directory named 'itemName'. 'itemName' 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 vshDeleteFile(const char *deleteFile)
```

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

```
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 vshRenameFile(const char *srcFile, const char *destFile)
```

Rename (move) the file specified by the name 'srcFile' to the destination 'destFile'. Both filenames must be absolute pathnames, 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.

```
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 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.