# SCI Kernel Documentation

Author: Jeff Stephenson
4 April 1988

Revision by: David Slayback
25 July 1989

Table of Contents

# SCI Overview

This document details the capabilities of the Script Interpreter (sci) and the interface to those capabilities through the Script programming language. Before getting into the specific functions we'll take a look at how sci operates in order to provide a background for the function explanations.

When sci starts up, one of the first things that it does is set up two memory spaces for memory management. One is the heap, and consists of an area smaller than 64K which can be addressed by 16 bit offsets. This is used for script code, variables, objects, lists, and other relatively small things which we wish to access quickly. When allocating memory from the heap, a pointer is returned, which is just the offset of the allocated memory within the heap. Things which exist in the heap are at fixed locations -- they are not affected by garbage collection.

The other memory space, hunk space, is much larger (128K or more, depending on the target machine) and is used to hold resources: views, pictures, text, background save areas, and anything else to which we do not need the access speed of 16 bit pointers. When a resource is loaded or allocated, sci returns a handle, which is a 16 bit pointer to a location in the heap which contains a 32 bit pointer to the resource. Since all resource access is through a handle, sci can do garbage collection in hunk space -- when there is not enough memory to load a resource, sci will rearrange the currently loaded resources in an attempt to create a large enough block of memory for the new resource. If the garbage collection is unable to create enough memory for the resource, sci will begin purging the least recently used resources from hunk space until there is enough room. Purging resources creates no problem, as any attempt to access a resource which is not in memory will cause it to be loaded. With garbage collection and automatic resource purging/loading, it is virtually impossible to run out of memory -- the worst condition which is likely to occur is constant disk access.

Once sci has initialized the memory management, graphics, etc., it turns control over to the Script pseudo-machine (pmachine), which loads script 0 and starts executing pseudo-code (pcode) with entry 0 in that script. From this point on, the pmachine executes the pcode produced by the sc compiler, making calls to the kernel routines in the interpreter and passing messages to objects to do things such as draw pictures, animate characters, etc.

Interfaces to the kernel are in kernel.sh. In the following descriptions, remember that any of the arguments can be an arbitrarily complex expression (including for example another kernel function call).

Many of the functions described here are never called directly, but are hidden in object methods (described in Script Classes for Adventure Games). These calls are given here for completeness and in case they are needed for writing new classes.

# Resource Functions

Resources are the various components of a game: views, pictures, scripts, etc. and, except for the MEMORY resource, reside on disk, waiting to be loaded when needed. Resources are handled automatically by the kernel, but several functions are provided to allow the programmer a bit more control over how the handling is done.

(Load resType resID)
Loads a resource of type resType and ID (number) resID into hunk space. While any attempt to use a resource which is not loaded will cause the resource to be loaded, it is not generally a good idea to rely on this as it may cause a disk-access delay in the midst of a crucial animation sequence. It is best to do an explicit Load of all resources during the initialization phase of a room to insure that everything is in memory. The resource types are:

```
VIEW
PICTURE
SCRIPT
TEXT
SOUND
MEMORY (used internally by the kernel)
VOCAB (used internally by the kernel)
FONT
CURSOR
```

Example: (Load SCRIPT forest) loads the script whose number is in the variable forest into memory.

(UnLoad resType resID)
Purges the resource of type resType and ID (number) resID from hunk space. This is not normally needed, as the kernel will automatically purge the least recently used resource when it doesn't have enough memory to load a new resource. This function is provided for a situation in which it is advantageous to override this automatic purging. Such a situation is unlikely.

Example: (UnLoad SOUND EXPLOSION) purges the sound whose number is EXPLOSION from memory.

(ScriptID script [entry])
Return the object ID for the object which at entry number entry in the publics table of script number script. This will load the script if it is not already in memory. If entry is not present, this returns the ID of entry number 0 of script.

(DisposeScript script)
While resources loaded into hunk space are automatically disposed of when no longer used, when a script is loaded it is loaded into the heap and remains there until disposed of by DisposeScript. This disposes of script number script and should be called when you no longer need the script. This is especially useful for discarding classes and regions that no longer need to be in memory.

(FlushResources roomNum)
If we are tracking resource usage (if run with 'sci -u'), set the internal room number of the room to roomNum, then flush all unlocked resources.

# List Functions

Lists play a crucial role in Script code, being the basic component out of which such classes as Collection, List, and Set are built. The kernel list (or kList, as it is known), is simply a block of memory with pointers to the first and last kNodes (kernel nodes) of the list. The kNodes are doubly linked, each having a pointer to both the previous and next node in the list. The nodes also have a key and a value associated with them. The value is the item being stored in the list (an integer, an object ID, etc.) and the key is a means for looking up the value.

(NewList)
Returns a pointer to a new kList with no elements.

(DisposeList kList)
Dispose of a kList. This involves disposing of all kNodes in the list (but not any objects pointed to by the kNode value), then disposing of the kList header itself.

(NewNode key value)
Create a new kNode, set key and value for the kNode, then return a pointer to the new kNode.

(FirstNode kList)
Returns a pointer to the first node in kList, or NULL if kList is empty.

(LastNode kList)
Returns a pointer to the last node in kList, or NULL if kList is empty.

(EmptyList kList)
Returns TRUE if kList is empty, FALSE otherwise.

(NextNode kNode)
Returns the node which follows kNode in a list or NULL if kNode is the last node in the list. Note that this returns the kNode, not the value of that node.

(PrevNode kNode)
Returns the node which precedes kNode in a list or NULL if kNode is the first node in the list. Note that this returns the kNode, not the value of that node.

(NodeValue kNode)
Return the node value of kNode.

(AddAfter kList kNode aNode [key])
Add aNode to kList immediately following kNode (which had better be an element of kList). If key is present, set the key of aNode to its value. Returns aNode.

(AddToFront kList kNode [key])
Add kNode to the front of kList. If key is present, set the key of kNode to its value. Returns kNode.

(AddToEnd kList kNode [key])
Add kNode to the end of kList. If key is present, set the key of kNode to its value. Returns kNode.

(FindKey kList key)
Return the first kNode in kList to have key as its key, or NULL if there is no node with the key.

(DeleteKey kList key)
Delete the first kNode in kList which has key as its key. Return TRUE if a node was deleted, FALSE if no node with the given key was found.

# Object Functions

These functions are low-level functions for manipulating and getting information about objects.

(Clone instance/class)
Return the object ID of a copy of instance/class. This copy has the same property values and the same methods as instance/class. The properties can be changed, the methods cannot.

(DisposeClone object)
Dispose of object if it was created with the Clone function, leave it alone otherwise. This does not dispose of any objects which may have their IDs in a property of object -- you must do that before calling DisposeClone.

(IsObject object)
Returns TRUE if object is an object or class, FALSE otherwise. Useful for testing to see that something is an object before sending a message to it in a situation in which you can't be guaranteed that a value is an object ID.

(RespondsTo object selector)
Returns TRUE if selector is a valid selector for object, i.e. if selector is the name of a property or method of object.

(Wait n)
Wait until n timer ticks (1/60th of a second) have passed since the last call to Wait. This is used to keep animation running at a constant speed -- each pass through the main loop ends with a call to Wait, so the main loop is executed at most once in a given time interval. The standard value of n is 6, leading to animation every 1/10th of a second. If more than n ticks have occurred since the last call to Wait, it returns immediately. The return value of Wait is the number of ticks more than n since the last call.

(GetTime [realTime])
With no arguments, returns the low word of the number of ticks (1/60th of a second) since the game was booted. With an argument, returns real system time in the format:

HHHH/MMMMMM/SSSSSS

# String Functions

Strings in Script are kept in arrays, with two characters per array element. Thus, when allocating space for a string of 40 characters, you only need to allocate an array of 20 elements.

(ReadNumber string)
Returns the integer value represented by string.

(Format stringPtr formatStr arg1 arg2 ...)
Formats a string in the storage space pointed to by stringPtr based on the format string formatStr and the arguments arg1, arg2, etc. Formatting commands embedded in formatStr consist of:

%[justification][field_width]conversion_char

[justification]: if a minus sign is present, then the the string representing the argument is to be right justified (rather than the default of left justified) in its field.

[field_width]: number indicating the width of the field (in characters) in which the argument is to be printed.

conversion_char:

| | |
|---|---|
| d | Print the corresponding argument as a signed decimal integer. |
| u | Print the corresponding argument as an unsigned decimal integer. |
| x | Print the corresponding argument as a hexadecimal number. |
| c | The corresponding argument is taken to be the ASCII representation of a character, which is printed. |
| s | The corresponding argument is assumed to be a pointer to a null terminated string, which is printed. |

--- Examples:
If we have declared str somewhere as [str 40] (an 80 character string), then

```
(Format @str "x:%4d y:%-4d" 23 45) -> "x:23   y:  45"

(Format @str "This is a %s." "test") -> "This is a test."
```

(GetFarText module entryNumber buffer)
Gets text from a script text file and copies it into buffer. module points to the script #, and entryNumber points to the text number within that file to use.

(StrCmp str1 str2)
Compares the null-terminated strings pointed to by str1 and str2. Returns 0 if the strings are the same, 1 if str1 is greater than str2 (i.e. if the first character of str1 which does not match the corresponding character of str2 is greater than the character in str2), and -1 if str1 is less than str2.

(StrLen str)
Returns the number of characters in the null terminated string pointed to by str.

(StrCpy str1 str2)
Copies the string pointed to by str2 to the storage pointed to by str1. There had better be enough room in str1's storage to hold str2 -- there is no checking.

(StrCat str1 str2)
Concatenates str2 to the end of str1. Str1 had better have enough storage.

(StrEnd str)
Returns a pointer to the NULL which terminates str. This is useful for Formatting a string on the end of another, rather than Formatting to a temporary string and then using StrCat.

(StrAt string position [char])
   StrAt returns the character at 'position' in 'string'.  If the optional
   'char' is specified, it replaces the character with 'char', returning
   the old contents.  Use this rather than the mask-and-shift method
   since byte order in a word will vary between machines.

# Picture Functions

(DrawPic picNum [showStyle] [clearPic] [pallete])
Clear the background screen, then draw picture number picNum in it. The picture will not be brought to the screen until the first Animate call following the DrawPic. To bring the picture to the screen immediately, call (Animate NULL).

The optional showStyle specifies the manner in which the kernel will bring the picture to the screen -- current possibilities are horizontal, vertical, left, and right wipe, horizontal and vertical shutter, iris in and out, dissolve, or just plain. See system.sh for the style constants.

If clearPic is FALSE, the current picture will overlay onto the existing picture, otherwise the default behavior will happen -- the previous picture will be cleared before being drawn.

palette, ranging from 0 - 3, will specify which of the 4 palettes to use to draw the picture. The default is 0.

(Show what)
Displays a given screen (visual, priority, or control) based on the value of what. This can be used for debugging to see why an actor is not able to enter a given area or why priorities aren't working properly. The values of what are one of

| | |
|---|---|
| VMAP | (visual screen -- you're generally displaying this) |
| PMAP | (priority screen -- objects will have their priorities displayed, and animation will continue on this screen) |
| CMAP | (control screen -- animation is stopped pending a keystroke when this screen is displayed, since you wont be able to see the Actors) |

(PicNotValid [value])
Returns TRUE if the picture window is actually shown on the screen, FALSE if the screen needs to be updated. If [value] is passed, sets the status to [value].

(ShakeScreen num [dir])
num is how many times to shake the screen, and dir is the direction to do the shaking: 1=down(default), 2=right, 3=down/right

(CoordPri yCoord)
Returns the priority that corresponds to yCoord on the priority map.

# Save/Restore Game Functions

(SaveGame name num @comment version)
   Saves the whole heap, including scripts loaded and all global,
   local, and temporary variables. Name is a string of the current
   game (ie. King's Quest#1 = KQ1), num is the number of the save game,
   comment is a pointer to a string that will be saved for later
   lookup, and version is the current version of the SCI
   interpreter. The game is saved under <name>SG.<num> and the file
   <name>SG.DIR contains the comment and the number for the given
   save game.

(RestoreGame name num version)
   RestoreGame restores the heap in the saved game, and then starts
   the game from where it had been saved. If the current running SCI
   version does not match the restored game's one, then a message
   will be printed and the game will not be restored.

(RestartGame)
   This function resets the system to the state it had at the beginning of
   the game, allowing the user to restart the game without rebooting it.

(GameIsRestarting [flag])
   If flag is present will set the GameIsRestarting internal flag
   to the given value, TRUE or FALSE. With no arguments, the function
   returns the value of the internal flag.
   The internal flag is set by RestartGame, and reset every time in
   the doit: of theGame.

(GetSaveDir)
   Return a string pointing to the current save directory.

(CheckSaveGame gameName fileNum version)
   Check the save game file "<gameName>SG.<fileNum>" to see if the
   versions match and so it can be restored.

# Animation Functions

(Animate cast [doCastDoit])
Cast is a kList of members of the cast (Actors, Props, and Views which are on the screen). Animate updates the on-screen views and positions of all members of the cast to correspond to the current state of their properties. doCastDoit can be either the default of TRUE meaning that for each member of the cast, the doit: method will be called, or FALSE, which will allow an animate cycle without the doits called.

If cast is 0, then Animate will dispose lastCast (internal kernel knowledge of the cast during the previous animation cycle) and redraw the picture, if needed.

If a picture has been drawn since the last Animate, the entire screen is updated. Certain bits in the signal properties of the objects allow an object to be erased and removed from the cast or tell Animate to leave the object in the cast but not to update it (in order to gain speed when an object isn't changing).

(CanBeHere actor castElements)
Checks to see that an Actor can be in a certain position. First, CanBeHere checks all pixels in an actor's base rectangle to see if any are on pixels which have the controls specified by actor's illegalBits property. Then, it checks to see if the actor's base rectangle intersects with any other of the casts base rectangle. If either of these are true, CanBeHere returns FALSE, otherwise TRUE.

(InitBresen motion)
Initialize internal state of a motion class for a modified Bresenham line.

(DoBresen motion)
Move an actor one step along the calculated path.

(DoAvoider avoider client motion [skipFactor])
Use avoider to move the client in the given motion, trying to avoid all objects and ignore all illegal controls. If skipFactor is given, the avoider will try to go skipFactor steps in a given move.

(SetJump actor deltaX deltaY gravY)
Compute the initial xStep for a motion of class Jump based on the x and y differences of the start and end points and the force of gravity. This was downcoded from Script to use longs to avoid overflow errors.
For most motion (increasing y or x motion comparable to or greater than y motion), we pick equal x & y velocities. For motion which is mainly upward, we pick a y velocity which is n times that of x.

(BaseSetter actor)
Set the actor's base rectangle (brTop, brLeft, brBottom, brRight) based on the y-value of the actor.

(DirLoop actor heading)
Set the loop of the actor based on the actors heading.

(NumLoops actor)
Returns the number of loops in the current view of actor.

(NumCels actor)
Returns the number of cels in the current loop and view of actor.

```
(SetNowSeen actor)
   Set the nowSeen rectangle of actor based on actor's current cel.

(CelWide view loop cel)
   Return the width (in pixels) of cel cel of loop loop of view view.

(CelHigh view loop cel)
   Return the height (in pixels) of cel cel of loop loop of view view.

(OnControl mapType x y [rx ly])
  Return a bit-mapped word which represents the control in
  mapType, where mapType = { VMAP, PMAP, CMAP}, screen at the
  point (x,y).  If the optional rx, ly are specified, the
  word has the bit set for each control which is within the rectangle
  (x, y) - (rx, ly).

(DrawCel view loop cel left top priority)
   Draw cel cel of loop loop of view view.  Put the upper left corner of
   the cel at (left, top).  The cel should be at priority priority.

(AddToPic picViewList)
   Will sort the given picViewList based on the y and z properties,
   and then will draw list of picViews to the picture.
```

# Graphic Screen Functions

The following kernel functions are direct calls to the graphic code used to draw the pictures and views.

mapSet is used below to refer to a word that has its bits set for a combination of showing on the Visual, Priority, and/or Control screen.

(Graph GLoadBits bitMap)
Load the given bitMap number, and display on screen. This function does NOT work for 16-color SCI.

(Graph GDetect)
Return the number of colors supported by the video hardware. EGA=16, CGA B/W=2, CGA Color=4, VGA=256.

(Graph GSetPalette paletteNum)
Not implemented yet -- for 256-color SCI.

;; changed 4-2-90 j.m.h. /m.w.
(Graph GDrawLine t l b r vColor pColor cColor)
Draw a line from (l, t) to (r, b) in the the given colors. A color value of -1 specifies that the line should not be drawn into that map.

(Graph GFillArea x y mapSet [vColor] [pColor [cColor]])
Do a fill at (x,y) in the mapSet screens in the given colors.

(Graph GDrawBrush x y size randomSeed mapSet [vColor] [pColor [cColor]])
Not implemented yet.

(Graph GSaveBits top left bottom right mapSet saveID)
Return a pointer to the saved bits of the rectangle (top,left, bottom, right) on the given screen(s) in mapset.

(Graph GRestoreBits saveID)
Restore to the screen the bits that are pointed to by saveID.

(Graph GEraseRect top left bottom right)
Draws visual in background color.

(Graph GPaintRect top left bottom right)
Draws visual in foreground color.

(Graph GFillRect top left bottom right mapset vColor pColor cColor)
Fill a given rectangle(top,left,bottom,right) in the given screen(s) and color(s).

(Graph GShowBits top left bottom right mapSet)
Not implemented yet.

(Graph GReAnimate top left bottom right)
This has the same affect as a ShowBits, BUT re-animates the cast members that are inside the shown rectangle.

(Graph GInitPri horizon foreground)
This sets the point at which animated characters change priority as they move up and down the screen. The horizon is the coordinate in the picture where their priority is 0. The foreground is the point where the priority will be 14. These values can be set in the picture editor, but are not saved. Default is hor=42, fore=190.

# Input Functions

(Said saidStr)
Checks to see if the parsed input sentence matches the input specified by saidStr. Returns TRUE if the input matched saidStr, FALSE otherwise.

(HaveMouse)
Returns TRUE if the user has a mouse driver installed, FALSE otherwise.

(SetCursor form showOn [x] [y])
Will set the cursor to the one found in the file "cursor.form" and show the cursor if showOn is TRUE. The optional x & y will position the cursor to the given position.

(GetEvent eventMask event)
Checks the input buffer for an input event of type specified by eventMask. Returns FALSE if there are none. If an event exists, it fills in the event record of the event instance whose ID is in event and returns TRUE.

The types of events which may be specified in eventMask are:

| | |
|---|---|
| mouseDown | a mouse button was pressed |
| mouseUp | a mouse button was released |
| keyDown | a key was pressed |
| keyUp | a key was released |
| menuStart | the menu request key was hit |
| menuHit | a menu item was selected |
| direction | a direction event was received |
| saidEvent | a Said statement |
| joyDown | a joystick button was pressed |
| joyUp | a joystick button was released |

These event types may be 'or'ed together to request multiple event types. The symbolic value 'allEvents' requests any event type.

(GlobalToLocal event)
Convert the coordinates in the event object event from global (screen) to local (window) coordinates. Event coordinates are always returned in global coordinates, so this call is necessary to convert to the coordinates within the current window.

(LocalToGlobal event)
The reverse of GlobalToLocal -- converts the coordinates in event from the local window coordinates to global screen coordinates.

(Parse stringPtr)
Parses the string pointed to by stringPtr and returns TRUE if the kernel could parse the string, FALSE otherwise. The kernel can parse the string if all the words in the string are in the game's vocabulary (the file vocab.000) and the sentence structure is one recognized by the kernel's grammar.

(SetSynonyms regionList)
Lets the kernel known about the synonyms in each region.

(MapKeyToDir flag)
If flag is TRUE(default), direction keys are mapped to direction events, otherwise if FALSE, direction keys are mapped to keyDown events.

# Menu Functions

The menu bar is the line at the top of the screen which contains the names of each of the menus. A menu is the drop-down list of menu items which can be selected.

(DrawMenuBar menuList)
Draw the menu bar represented by the kList menuList on the top line of the screen in the system font (font 0).

(MenuSelect menuList blocks)
Drop down the first (leftmost) menu in the menu bar represented by menuList and let the user select an item from the menu bar using cursor keys. If the user presses ESC, return FALSE; if the user presses ENTER, return the object ID of the item selected.

(DrawStatus str [foreground [background]])
Replace the menu bar with a status line which consists of the string str. If str is 0, discard the status line, showing the menu bar once again. (Note that the user can still activate the menus by pressing the mouse button with the cursor on the status line or by pressing the menu selection key.)
The foreground and background colors default to black on white. Note that these colors are indexes into the palette, so vRED is unlikely to be actually red (it will select the 4th element of the current palette).

(AddMenu menuTitle menuSelections)
Add a menu item that has menuTitle, which is a near string that contains the message that will be on the menu bar.
menuSelections is a near string that contains the available items that will "drop down" if the menu is selected.
The menuSelections is built by putting together the items, seperated by a ":" as follows -

```
{ <name1><functions>:<name2><functions>...}
```

The following functions are possible :

```
= sets a menu item's starting value
! makes the item non-selectable
` denotes the following character as the key for the menu
```

An example of the use of AddMenu is:

```
(AddMenu { File }
         { Save Game`#5:--!:Restore Game`#6:Speed`^s=6}
)
```

(SetMenu itemName selector value [selector value ...])
where itemName is the name by which you refer to your menu item and the the available selectors are

```
#p_said: newSaidSpec     change the said spec for the menu item
#p_text: newText         change the text displayed in the menu
#p_key: newKey           change the key which selects the menu item
#p_state: newState       = dActive to enable menu item
                         = 0 to disable menu item
#p_value: newValue       change the value to return when selected
```

(GetMenu itemName selector)
Returns the current value of a menu item corresponding to the selectors listed above.

# Window and Text Functions

These functions deal with the main picture window, dialog windows, and writing text to those windows.

(GetPort)
Returns the current grafPort.

(SetPort grafPort)
Sets the current grafPort to grafPort. If grafPort=0, then the port is set to the window manager port, which is the full screen picture with (0,0) right below the menu bar.

(NewWindow top left bottom right title type priority color back)
Opens a window with the given coordinates, and returns a handle to the new window.

(DisposeWindow window)
Close the given window.

(Display text [at: x y] [font: f] [color: c] [back: b] [style: s] [width: w])
Writes text to the current grafPort. The optional parameters are:

p_at: x y — Position the upper left corner of the first character of text at coordinates x, y in the grafPort.

p_font: f — Write the text in font f.

p_color: c — Set the foreground color of the characters to c. The characters will be ORed into the picture unless "back:" (see below) is specified.

p_back: b — Set the background color of the characters to b. Additionally makes the characters OPAQUE.

p_style: s — Set the style of the characters to s. s may be TPLAIN (plain text), TDIMMED (dimmed text) or TBOLD (bold text).

p_width: w — Sets the width of the displayed line. The text will wrap at the width that you specify. If this parameter is not passed, the text will NOT wrap, and long lines will go off the edge of the screen.

p_mode: j — Set justification of text. Left aligned is default. (teJustLeft, teJustCenter, or teJustRight)

p_save: — Saves the background under the window and returns a value that relates to the saved area. Use (= var (Display "Lots o text" save:)

p_restore: var — Restores the background that was saved. Use (Display "" p_restore: var)

(TextSize rectPtr text font width)
Fills in the rectangle (an array of four elements) which is pointed to by rectPtr with the bounding coordinates of the box which will hold text printed in the font font. If width is non-zero, it is the maximum width of the rectangle.

```
(DrawControl item)
   Draw a control object within the given port
   item can be dSelector, dButton, dText, dIcon, or dEdit

(HiLiteControl item)
   Hilight the control item if not selected, or unhilight the item if
   selected.

(EditControl item event)
   Bring up an edit box for a the item object that has type=dEdit,
   and store the message passed through to event.
```

# Sound Functions

(DoSound InitSound sound)
   Initialize the sound. Will set handle property of object to the
   internal sound node.

(DoSound PlaySound sound)
   Plays the sound, if no other higher priority sounds are playing.

(DoSound NextSound)
   Not implemented yet.

(DoSound KillSound handle)
   Kill the sound given in handle.

(DoSound SoundOn [soundFlag])
   If soundFlag is present, it will either turn on sound output if
   soundFlag is TRUE or turn off sound output if soundFlag is FALSE.
   If soundFlag is not present, it will returns TRUE if sound output
   is on, FALSE otherwise.

(DoSound StopSound handle)
   Stop the sound specified in handle from playing.

(DoSound PauseSound value)
   If value is TRUE, pause all the active sounds and set state to
   SND_BLOCKED.. If value is FALSE, then unpause all blocked sounds.

(DoSound RestoreSound)
   Loads all sounds in sound list and starts playing the one that is
   active that has the highest priority.

(DoSound ChangeVolume vol)
   Sets the volume to vol, where vol can be between 0 and 100.

(DoSound ChangeSndState soundObj)
   Change the state of the internal sound node to correspond to the
   values in soundObj.

(DoSound FadeSound handle)
   On systems with volume control, fade away the sound given in handle.
   Otherwise, a StopSound is implemented in the music driver.

(DoSound NumVoices)
   Returns the number of voices in the sound hardware.

# Arithmetic Functions

(Random min max)
   Returns a random number n such that 0 < min <= n <= max.

(Abs number)
   Returns the absolute value of the signed integer number.

(Sqrt number)
   Returns the square root of number, which is assumed to be unsigned.

(GetDistance x1 y1 x2 y2 [perspective])
   Returns the distance between the two points determined by (x1, y1) and (x2, y2). perspective is the users point of view of the room in degrees away from the vertical along the y axis, which makes each y-pixel represent a greater distance than an x-pixel.

(GetAngle x1 y1 x2 y2)
   Returns the angle between the two points determined by (x1, y1) and (x2, y2). The angle is measured in degrees and is between 0 and 359 degrees.

(SinMult angle num)
   Returns num multiplied by the sine of angle.

(CosMult angle num)
   Returns num multiplied by the cosine of angle.

(SinDiv angle num)
   Returns num divided by the sine of angle.

(CosDiv angle num)
   Returns num divided by the cosine of angle.

# File Functions

These functions allow access to some of the MS-DOS file functions. They are best accessed through the File class.

(FileIO fileOpen filename [mode])
Opens the file whose name is filename and returns a handle to it. The optional parameter mode may be either fAppend, in which case the file pointer will be positioned at the end of the file for appending, or fTrunc, in which case the file will be truncated to zero length, or FRead in which case the file pointer will be positioned at the beginning of the file. If mode is not specified, fAppend is assumed. If there is an error in opening the file, a value of -1 is returned, else the file handle.

(FileIO fileFPuts handle string)
Write the text pointed to by string to the file whose handle is handle. Returns number of bytes successfully written.

(FileIO fileFGets string len handle)
Get some text of max length len from the file whose handle is handle and stores it in string. Returns string.

(FileIO fileWrite handle address length)
Write the length bytes from address and return the number of bytes successfully written.

(FileIO fileRead handle address length)
Read length bytes into address and return the number of bytes successfully read.

(FileIO fileSeek handle offset mode)
Changes the position in the file where the next read or write will occur. The new position is specified by means of the mode and offset. If the mode is fileSeekCur the offset is relative to the beginning of the file. If the mode is fileSeekCur the offset is relative to the current position. If the mode is fileSeekEnd the offset is relative to the end of the file. The offset can be negative (for fileSeekCur and fileSeekEnd modes). The new file position is returned.

(FileIO fileClose handle)
Close the file whose handle is handle.

(FileIO fileUnlink filename)
Delete filename, returning 0 if unsuccessful.

(FileIO fileFindFirst mask @fileName attribute)
Find the first file matching mask and put its name into fileName. See a DOS reference for attribute usage (it's somewhat warped).

(FileIO fileFindNext @fileName)
Continue the search started by fileFindFirst.

(FileIO fileExists @fileName)
Determine if fileName exists.

(GetSaveFiles gameName filenames nums)
Read the save-game directory for all files matching <gameName>*.num, putting file descriptions in the array pointed to by filenames, the file numbers in the array pointed to by nums. Returns the number of save games found.

(GetCWD pathNamePtr)
Get the current working directory and store it in pathNamePtr.

(CheckFreeSpace pathName)
See if there is enough free space on the disk in pathName to save the current heap.

(ValidPath pathNamePtr)
Return TRUE if the passed path is valid, FALSE otherwise.

```
(DeviceInfo GetDevice @path @device)
   Puts the string describing the device component of 'path'
   into the string pointed to by 'device'.  Thus, if
   path = "g:/games/kq4/sci", device = "g:".  If there is no
   device component in 'path', puts the current device in 'device'.

(DeviceInfo CurDevice @device)
  Puts the string describing the current device in 'device'.

(DeviceInfo SameDevice @dev1 @dev2)
  Returns TRUE if the strings pointed to by 'dev1' and 'dev2' are
  the same physical device, FALSE otherwise.

(DeviceInfo DevRemovable @device)
  Returns TRUE if 'device' is removable, FALSE otherwise.
```

# Debugging Functions

There a number of functions designed for debugging Script programs by providing information about the state of the program.

(SetDebug)
Pop up the debugging window. At the top of the window, in red, is the name (if any) of the object self. Below it is the op-code about to be executed. In columns on the right side of the window are the top five values on the stack and the top five parameters. On the left are the contents of the accumulator (acc), the address of the next instruction (the instruction pointer, or ip), and the address of the top of the stack (the stack pointer, or sp).

A number of instructions may be issued while in the debugger (Note that the debugger is case-sensitive, ie. Q != q):

<shft><shft>-
Put the debugging window away. This also can be used to pop the debugging window up while the program is running.

q
Quit. Exit to DOS. Using <shft><shft>- to pop up the debugger and q to quit will generally get you out of the program even if your code is broken.

s
Toggle the send stack on/off.

Enter
Step to the next instruction, tracing into the procedure or method referenced by a call or send instruction.

Tab
Step to the next instruction treating a call or a send as an indivisible instruction (don't trace into them).

tn
Display the value of temporary variable number n.

ln
Display the value of local variable number n.

gn
Display the value of global variable number n.

i
Open an inspector window, allowing you to inspect the values of the properties of objects. See InspectObj below.

o
Look at all objects.

O
Look at all objects with their hex addresses given. The actual address to inspect is <addr>+$0006.

a
Look at object in the accumulator.

c
Look at current object on top of send stack.

(InspectObj object)
Open an inspector window on object. This displays the property names and values for the object. Typing 'i' when this is displayed prompts for a property name, whose value will be displayed either as a number, string, or another object depending on what it is. If another object, 'i' can be typed again to inspect its properties, and so on until the interpreter runs out of stack.

(ShowSends)
   Show the current send stack.  This allows you to see how you got where you are.  Entries in the display are of the form (object selector:) where object is the object to which a message whose selector was selector: was sent.  The top line in the display is the most recent send, the line below that is the send to the method which made that send, and so on to the bottom line, which is the initial send from the main loop in the base script.

(ShowObjs withID)
   Display all static and dynamic objects which are currently in the heap. If withID is TRUE, show the object IDs as well.

(ShowFree)
   Displays the free memory blocks in the heap in the form number-of-bytes@address.

(MemoryInfo LargestPtr)
   Returns the size (in bytes) of the largest block of memory available in the heap.

(MemoryInfo LargestHandle)
   Return the size (in bytes) of the largest hunk available in hunk space. If the largest available hunk is greater than 64K, returns 64K.

(MemoryInfo FreeHeap)
   Return the amount of memory (in bytes) which is free in the heap.

(MemoryInfo FreeHunk)
   Return the amount of memory (in paragraphs, or 16 byte blocks) which is free in hunk space.

(StackUsage MStackSize)
   Return the stack size of the PMachine stack.

(StackUsage MStackMax)
   Return the maximum stack size reached so far of the PMachine stack.

(StackUsage MStackCur)
   Return the current stack size of the PMachine stack.

(StackUsage PStackSize)
   Return the stack size of the Processor stack.

(StackUsage PStackMax)
   Return the maximum stack size reached so far of the Processor stack.

(StackUsage PStackCur)
   Return the current stack size of the Processor stack.

Index