The Sierra Creative Interpreter

Lars Skovlund, Christoph Reichenbach, Ravi Iyengar,
Rickard Lind, Vladimir Gneushev, Petr Vyhnak,
Dark Minister, Francois Boyer, Carl Muckenhoupt

November 21, 2013

Legal notice

Abstract

This book describes the Sierra Creative Interpreter, versions 0.xxx and 1.xxx to the extent known to the general public, as well as the FreeSCI interpreter for those games. Please contact the author if you find that anything is being described incorrectly or missing.

NOTE: This version of the documentation is incomplete and covers only some parts of SCI0.

Preface

Throughout the documentation, the term SCI will be used to describe the original Sierra Creative Interpreter, in any version. SCI0 will refer to all games using the SCI version 0.xxx, except for those games who use the 'in-between' game engine referred to as SCI01 (such as Quest for Glory 2). SCI1 will refer to the interpreter version 1.xxx. FreeSCI will refer specifically to either implementation details of the FreeSCI engine or to extensions of the original SCI engine specific to FreeSCI.

I would like to take this opportunity to thank the members of the FreeSCI and SCI Decoding Projects and their supporters, as well as Carl Muckenhoupt, who took the first steps of SCI decoding, for their valuable help and support.

Please note that some of the text contributions have been cut, reformatted or slightly modified in an attempt to improve the general quality of this document.

Chapter 1

Introduction

1.1 The basics

The Sierra Creative Interpreter is a stack-based virtual machine (”P-Machine”). In addition to its roughly 125 basic opcodes, it provides a set of extended functions for displaying graphics, playing sound, receiving input, writing and reading data to and from the hard disk, and handling complex arithmetical and logical functions. In version 0.xxx of the interpreter, Sierra split the game data into nine different types of information:

- *script data*: SCI scripts and local data
- *vocab data*: Parser data and debug information
- *patch data*: Information pertaining to specific audio output devices
- *sound data*: MIDI music tracks
- *cursor data*: Mouse pointer shapes
- *view data*: Sets of sets of image and hotspot information
- *pic data*: Background images and metadata
- *font data*: Bitmap fonts
- *text data*: Plain text information

Each game may contain up to 1000 different elements of each data type; these elements are referred to as ”resources”. The index numbers of the various resources need not be in sequence; they are usually assigned arbitrarily. [1]

1.2 Resource storage

Individual resources can be stored in one of two ways: Either in resource files (which, surprisingly, are called something like ”resource.000” or ”resource.001”), or in external patch files (not to be confused with ”patch” resources). The external files are called something like ”pic.100” or ”script.000”, and they take precedence over data from resource files.

There is also a file called ”resource.map”, which contains a lookup table for the individual resources, and another file, ”resource.cfg”, which contains configuration information; neither of those is used by FreeSCI.

Resource information stored in external patch files is not compressed and therefore easily readable. It is, however, preceeded by two bytes: The first byte contains the resource type ORed with 0x80, the purpose of the second byte is unknown (but it appears to be ignored by the original SCI version 0 engine).

As stated before, external patch files take precedence over resource resource files. Applying those external files as patches is an option since FreeSCI version 0.2.2.

[1] With several notable exceptions, such as script 0 and most vocab resources.

The resource files, however, are more complicated. Each of them contains a sequence of resources preceeded by a header; these resources may be compressed. It is, also, quite common to find resources shared by several resource files. The reason for this appears to be that that, back when hard disks were rare and hard to come by, the games had to be playable from floppy disks. To prevent unneccessary disk-jockeying, common stuff was placed in several resource files, each of which was then stored on one disk.

1.3 The individual resources: A summary

The resource types of SCI0 can be roughly grouped into four sets:

- Graphics (pic, view, font, cursor)
- Sound (patch, sound)
- Logic (script, vocab)
- Text

Text resources are nothing more than a series of ASCIIZ strings; but the other resources deserve further discussion.

1.3.1 Graphical resources summarized

The screen graphics are compromised of the four graphics resources. The background pictures are drawn using vector-oriented commands from at least one pic resource (several resources may be overlaid). The fact that vector graphics were used for SCI0 allows for several interesting picture quality improvements. Pic resources also include two additional "maps": The priority map, which marks parts of the pictures with a certain priority, so that other things with less priority can be fully or partially covered by them even if they are drawn at a later time, and the control map, which delimits the walking area and some special places used by the game logic. FreeSCI uses a fourth auxiliary map for during drawing time (this is a heritage from Carl Muckenhoupt's original code).

View resources contain most of the games' pixmaps (multi-color bitmaps). Each view contains a list of loops, and each loop contains a list of cels. The cels themselves contain the actual image information: RLE encoded pixmaps with transparency information, and relative offsets.

View resources are used for foreground images as well as for background images (for example, the "Spielburg" sign in QfG1 (EGA) is stored in a view resource and added to the background picture after it is drawn).

The cursor resource contains simple bitmaps for drawing the mouse pointer. It only allows for black, white, and transparent pixels in SCI0.

The fourth graphics resource is font data. It contains bitmapped fonts which are used to draw most of the text in the games. Text is used in one of four places: Text boxes, Text input fields, the title bar menu, and occasionally on-screen.

1.3.2 Sound resources summarized

SCI0 uses two types of resources for sound: Patch resources, and sound resources. Sound resources contain a rather simple header, and music data stored in a slightly modified version of the MIDI standard.

Patch resources contain device-dependant instrument mapping information for the instruments used in the sound resources. SCI0 sound resources do not adhere to the General Midi (GM) standard (which was, to my knowledge, written several years after the first SCI0 game was released), though later SCI versions may do so.

1.3.3 Logic resources summarized

Whenever the parser needs to look up a word, it looks for it in one of the vocab resources. This is not the sole purpose of the vocab resources, though; they provide information required by the debugger, including the help text for the debugger help menu and the names of the various SCI opcodes and kernel functions.

Script resources are the heart (or, rather, the brains) of the game. Consqeuently, they also are its most complex aspects, containing class and object information, local data, pointer relocation tables, and, of course, SCI bytecode.

To run the game, scripts are loaded on the SCI stack, their pointers are relocated appropriately, and their functions are executed by a virtual machine. They use a set of 0x7d opcodes, which may take either 8 or 16 bit parameters (so, effectively, there is twice the amount of commands). The functions may refer to global data, local temporary data, local function parameter data, or object data (selectors). They may, additionally, indirectly refer to "hunk" data, which is stored outside of the SCI heap. Since the whole design is object oriented, functions may re-use or overload the functions of their superclasses.

1.4 SCI01 extensions

SCI01 differs only in very few respects: It uses different compression algorithms (all of which are supported since FreeSCI 0.2.1), and a different type of sound resources, which may contain digitized sound effects (PCM data). The basic music data, however, still resembles MIDI data.

Also, scripts are split into two parts when loaded: A dynamic part, which resides in the heap as before, and a static part, which is stored externally to conserve heap space. [2]

1.5 SCI1 extensions

SCI1, which is not covered by FreeSCI at the moment, introduces new concepts like Palettes, scaled bitmap images and several new compression algorithms. In SCI1.0, the resource limit was first increased to 16383 [3] , and then to 65535 in SCI1. Because of the inherent limitations of the FAT file system the primary target OS of Sierra's SCI interpreter was limited to, patch file names were altered accordingly, with the resource number (not padded) before the dot and a three-letter resource ID behind it; examples are "0.scr" or "100.v56".

The complete list of suffixes is as follows:

80: v56: 256 color views
81: p56: 256 color background pictures
82: scr: Scripts (static data)
83: tex: Texts (apparently deprecated in favor of messages)
84: snd: Sound data (MIDI music)
86[4] :voc: Vocabulary (not used)
87: fon: Fonts
88: cur: Mouse cursors (deprecated in favor of v56-based cursors)
89: pat: Audio patch files
8a: bit: Bitmap files (purpose unknown)
8b: pal: 256 color palette files
8c: cda: CD Audio resources
8d: aud: Audio resources (probably sound effects)
8e: syn: Sync (purpose unknown)
8f: msg: Message resources: Text plus metadata
90: map: Map (purpose unknown)
91: hep: Heap resources: Dynamic script data

Apparently, the script resource split introduced in SCI01 was incorporated into the actual resource layout in SCI1.

1.6 Sierra SCI games

Paul David Doherty, Vladimir Gneushev

The listing here is almost certainly incomplete. Thanks to the information provided by Vladimir, game information now includes some features of certain versions the interpreter shipped with, they are listed below:

[2] The background for this is that heap space started running out in Quest for Glory 2. In order to compensate for this, changes were made to both the script library and the interpreter.

[3] This *appears* to be the limit- none of the SCI1.0 games I tested used resource numbers beyond 16383

Symbol	Meaning
Rn	Resource patches identified by name (script.256)
Re	Resource patches identified by extension (256.scr)
Dd	Built-in debugger
*D**	Interpreter binary shipped with debug symbols
Ss	Scripts consist of script resources only
Sh	Scripts use heap and script resources
Sc	Scripts use 'csc' resources

1.6.1 SCI0

Game name	ID	interpreter version	Parser	Map file ver.	More
Season's Greetings (1988)	DEMO	0.000.294	yes	0	Re Dd Ss
Leisure Suit Larry 2	LSL2	0.000.343	yes	0	Re Dd Ss
Police Quest 2	PQ2	0.000.395	yes	0	Re Dd Ss
Leisure Suit Larry 2	LSL2	0.000.409	yes	0	Re Dd Ss
Space Quest 3	SQ3	0.000.453	yes	0	Re Dd Ss
Police Quest 2	PQ2	0.000.490	yes	0	Re Dd Ss
King's Quest 4	KQ4	0.000.502	yes	0	Re Dd Ss
Fun Seeker's Guide	emc	0.000.506	yes	0	Re Dd Ss
Hoyle's Book of Games 1	cardGames	0.000.530	?	0	Re Dd Ss
Hero's Quest 1	HQ	0.000.566	yes	0	Re Dd Ss
Leisure Suit Larry 3	LSL3	0.000.572	yes	0	Re Dd Ss
Hoyle's Book of Games 2	solitare	0.000.572	yes	0	Re Dd Ss
Quest for Glory 1	Glory	0.000.629	yes	0	Re Dd Ss
The Colonel's Bequest	CB1	0.000.631	yes	0	Re Dd Ss
Codename: Iceman	iceMan	0.000.668	yes	0	Re Dd Ss
Hoyle's Book of Games 1	cardGames	0.000.685	?	0	Re Dd Ss
Conquest of Camelot	ARTHUR	0.000.685	yes	0	Re Dd Ss
Codename: Iceman	iceMan	0.000.685	yes	0	Re Dd Ss
Space Quest 3	SQ3	0.000.685	yes	0	Re Dd Ss

1.6.2 SCI01

Game name	ID	interpreter version	Parser	Map file ver.	More
King's Quest I	KQ1	S.old.010	yes	?	Dd Ss
Quest for Glory 2	Trial	1.000.072	yes	0	Re Ss
[Christmas greeting card 1990]	?	1.000.172	?	?	
[Christmas greeting card 1990]	?	1.000.174	?	?	
Space Quest 3 german	SQ3	?	bilingual	0	Re Dd Ss

1.6.3 SCI1

Game name	ID	interpreter version	Parser	Map file ver.	More
King's Quest 5	?	1.000.060	no	0	Re Ss
Leisure suit Larry 1 demo	?	1.000.084	no	0	Rn Ss
Conquest of the long bow	?	1.000.168	no	1	Rn Ss
Space Quest 1 demo	?	1.000.181	no	0	Rn Ss
Leisure Suit Larry 1 (VGA)	?	1.000.577	?	?	
King's Quest 5	?	1.000.784	?	?	
Space Quest 4	?	1.000.753	no	0	Re Ss

1.6.4 SCI1-T.A series

Game name	ID	interpreter version	Parser	Map file ver.	More
Police Quest 3 demo	?	T.A00.052	no	1	Rn Ss
Space Quest 1 (VGA)	?	T.A00.081	?	?	
Leisure suit Larry 5	?	T.A00.169	no	1	Rn Ss
Police Quest 3	?	T.A00.178	?	?	

1.6.5 SCI1 suspected forks

Game name	ID	interpreter version	Parser	Map file ver.	More
Jones in the Fast Lane	?	x.yyy.zzz	no	0	Re Dd Ss
Mixed-up mother goose demo win	?	x.yyy.zzz	no	0	Re Dd Ss
Eco Quest 1	?	1.ECO.013	no	1	Rn Ss
Mixed-up fairy tales demo	?	?????????	no	1	Rn Ss

1.6.6 SCI1.1

Game name	ID	interpreter version	Parser	Map file ver.	More
Eco Quest 1 demo	?	x.yyy.zzz	no	1	Rn D* Sh
Laura Bow 2 demo	?	x.yyy.zzz	no	1	Rn D* Sh
Hoyle's Book of Games 3	?	x.yyy.zzz	no	1	Rn D* Sh
Quest for Glory 3 demo	?	1.001.021	no	1	Rn Sh
LSL: Crazy Nick's Budget Picks	?	1.001.029	no	1	Rn D* Sh
Robin Hood's Games of Skill and Chance	?	1.001.029	no	1	Rn D* Sh
Parlor Games with Laura Bow	?	1.001.029	no	1	Rn D* Sh
King Graham's Board Game Challenge	?	1.001.029	no	1	Rn D* Sh
Leisure Suit Larry's Casino	?	1.001.029	no	1	Rn D* Sh
Roger Wilco's Spaced Out Game Pack	?	1.001.029	no	1	Rn D* Sh
Police Quest 1	?	1.001.029	no	1	Rn D* Sh
Quest for Glory 1 demo	?	1.001.029	no	1	Rn D* Sh
Quest for Glory 3	?	1.001.050	no	1	Rn Sh
Island of Dr. Brain 1	?	1.001.053	no	1	Rn Sh
Island of Dr. Brain 2	?	1.001.053	no	1	Rn Sh
Island of Dr. Brain 2 demo	?	1.001.053	no	1	Rn Sh
King's Quest 6	?	1.001.054	no	1	Rn Sh
King's Quest 6 demo	?	1.001.055	no	1	Rn Sh
Eco Quest 2 demo	?	1.001.055	no	1	Rn Sh
[Christmas greeting card 1992]	?	1.001.055	?	?	
Space Quest 4 windows	?	1.001.064	?	?	
Eco Quest 2	?	1.001.065	no	1	Rn Sh
Space Quest 5	?	1.001.068	no	1	Rn Sh
Space Quest 5 french	?	1.001.068	?	?	
Space Quest 5 german	?	1.001.068	?	?	
Freddy Pharkas demo	?	1.001.069	no	1	Rn Sh
Leisure suit Larry 6 dos+win	?	1.001.069	no	1	Rn D* Sh
Twisty history demo dos+win	?	1.001.069	no	1	Rn Sh
Twisty history demo dos+win	?	1.001.070	no	1	Rn Sh
Pepper's adventures in time	?	1.001.072	no	1	Rn Sh
Laura Bow 2	?	1.001.072	no	1	Rn Sh
Freddy Pharkas	?	1.cfs.081	no	1	Rn Sh
Gabriel Knight 1 demo	?	1.001.092	no	1	Rn Sh
Freddy Pharkas demo win	?	1.001.095	no	1	Rn Sh
Leisure suit Larry 6 dos+win	?	1.001.113	no	1	Rn D* Sh
King's Quest 6	?	1.cfs.158	?	?	
Laura Bow 2	?	2.000.274	no	1	Rn Sh
Quest for Glory 1 vga	?	L.rry.021	?	?	

Quest for Glory 3 german	?	L.rry.083	?	?	
Quest for Glory 1 vga	?	2.000.411	no	1	Rn Sh
Quest for Glory 4 demo	?	No number	no	1	Rn D* Sh

1.6.7 SCI32

Game name	ID	interpreter version	Parser	Map file ver.	More
Police Quest 4 floppy dos+win	?	?	?	?	
LightHouse	?	?	?	?	
LightHouse demo w9x (another)	?	?	?	?	
Space Quest 6	?	?	?	?	
Quest for Glory 4 floppy	?	2.000.000	?	?	
Quest for Glory 4 demo (another)	?	2.000.000	?	?	
Gabriel Knight 1	?	2.000.000	no	1	Rn Sh
Torin's passage dos+win	?	2.100.002	no	3	Rn Sh
Gabriel Knight 2 dos+win	?	2.100.002	no	3	Rn Sh
Police Quest: SWAT demo win	?	2.100.002	no	3	Rn Sh
King's Quest 7 win+w9x	?	2.100.002	no	3	Rn Sh
Phantasmagoria	?	2.100.002	?	?	
Quest for Glory 4 cd dos+win	?	2.100.002	no	3	Rn Sh
Shivers win	?	2.100.002	no	3	Rn Sh
Shivers demo win	?	2.100.002	no	3	Rn Sh
Phantasmagoria 2 w9x	?	3.000.000	no	3	Rn Sc
Leisure suit Larry 7 dos+w9x	?	3.000.000	no	3	Rn Sc
LightHouse demo w9x	?	3.000.000	no	3	Rn Sc
RAMA	?	3.000.000	no	3	Rn Sc
Shivers 2	?	3.000.000	no	3	Rn Sc

Chapter 2

Resource files

with significant contributions from Petr Vyhnak and Vladimir Gneushev

In order to allow games to be both distributeable and playable from several floppy disks, SCI was designed to support multi-volume data. The data itself could therefore be spread into separate files, with some of the more common resources present in more than one of them. The global index for these files was a "resource.map" file, which was read during startup and present on the same disk as the interpreter itself. This file contained a linear lookup table that mapped resource type/number tuples to a set of resource number/ offset tuples, which they could subsequently be read from.

2.1 SCI0 resources

2.1.1 resource.map

The SCI0 map file format is pretty simple: It consists of 6-byte entries, terminated by a six-tuple of 0xff values. The first 2 bytes, interpreted as little endian 16 bit integer, encode resource type (high 5 bits) and number (low 11 bits). The next 4 bytes are a 32 bit LE integer that contains the resource file number in the high 6 bits, and the absolute offset within the file in the low 26 bits. SCI0 performs a linear search to find the resource; however, multiple entries may match the search, since resources may be present more than once (the inverse mapping is not injective).

Early SCI01 (namely certain VGA games not using the later SCI1 resource.map format) uses a slight variation on this, which is almost identical: The first two bytes are unchanged, but the latter four only use the most significant 4 bits for storing the file number, and (consequently) 28 bits for the file offset.

2.1.2 resource.<nr>

SCI0 resource entries start with a four-tuple of little endian 16 bit words, which we will call (`id`, `comp_size`, `decomp_size`, `method`). `id` has the usual SCI0 semantics (high 5 are the resource type, low 11 are its number). `comp_size` and `decomp_size` are the size of the compressed and the decompressed resource, respectively. The compressed size actually starts counting at the record position of `decomp_size`, so it counts four bytes in addition to the actual content. `method`, finally, is the compression method used to store the data.

2.2 SCI1 resources

2.2.1 resource.map

The SCI1 resource.map starts with an array of 3-byte structures where the 1st byte is the resource type (0x80 ... 0x91) and next 2 bytes (interpreted as little-endian 16 bit integer) represent the absolute offset of the resource's lookup table (within resource.map). This first array is terminated by a 3-byte entry with has 0xFF as a type and the offset pointing to the first byte after the last resource type's lookup table. SCI1 first goes through this list to find the start of list for the correct resource type and remember this offset and the offset from the next entry to know where it ends. The resulting interval contains a sorted list of 6-byte structures, where the first LE 16 bit integer is the resource number, and the next LE 32 bit integer contains the resource file number in its high 4 bits and the absolute resource offset (in the

indicated resource file) in its low 28 bits. Because the list is sorted and its length is known, Sierra SCI can use binary search to locate the resource ID it is looking for.

2.2.2 resource.<nr>

Later versions of SCI1 changed the resource file structure slightly: The resource header now contains a byte describing the resource's type, and a four-tuple (`res_nr`, `comp_size`, `decomp_size`, `method`), where `comp_size`, `decomp_size`, and `method` have the same meanings as before (with the exception of `method` referring to different algorithms), while res_nr is simply the resource's number.

Rumor has it that late versions of SCI1 also stored the offsets shifted to the right by two bits (thus, all resources are always stored at word-aligned offsets in these games).

2.3 Decompression algorithms

The decompression algorithms used in SCI are as follows:

Table 2.1 SCI0 compression algorithms

method	algorithm
0	uncompressed
1	LZW
2	HUFFMAN

Table 2.2 SCI01 compression algorithms

method	algorithm
0	uncompressed
1	LZW
2	COMP3
3	HUFFMAN

Table 2.3 SCI1.0 compression algorithms

method	algorithm
0	uncompressed
1	LZW
2	COMP3
3	UNKNOWN-0
4	UNKNOWN-1

As reported by Vladimir Gneushev, SCI32 uses STACpack (as described in RFC 1974) explicitly, determining whether there is a need for compression by comparing the size of the compressed data block with that of the uncompressed.

2.3.1 Decompression algorithm LZW

The LZW algorithm itself, when used for compression or decompression in an apparatus (sic) designed for compression and decompression, has been patented by Unisys in Japan, Europe, and the United States. Fortunately, FreeSCI only needs LZW decompression, which means that it does not match the description of the apparatus as given above. (Further, patents on software are (at the time of this writing) not enforceable in Europe, where the FreeSCI implementation of the LZW decompressor was written).

WriteMe.

Table 2.4 SCI1.1 compression algorithms

method	algorithm
0	uncompressed
18	DCL-EXPLODE
19	DCL-EXPLODE
20	DCL-EXPLODE

2.3.2 Decompression algorithm HUFFMAN

This is an implementation of a simple huffman token decoder, which looks up tokens in a huffman tree. A *huffman tree* is a hollow binary search tree. This means that all inner nodes, usually including the root, are empty, and have two siblings. The tree's leaves contain the actual information.

```
FUNCTION get_next_bit(): Boolean;
/* This function reads the next bit from the input stream. Reading starts at the MSB.


FUNCTION get_next_byte(): Byte
VAR
    i: Integer;
    literal: Byte;
BEGIN
    literal := 0;
    FOR i := 0 to 7 DO
        literal := (literal << 1) | get_next_bit();
    OD
    RETURN literal;
END


FUNCTION get_next_char(nodelist : Array of Nodes, index : Integer): (Char, Boolean)
VAR
    left, right: Integer;
    literal : Char;
    node : Node;
BEGIN
    Node := nodelist[index];

    IF node.siblings == 0 THEN
        RETURN (node.value, False);
    ELSE BEGIN
       left := (node.siblings & 0xf0) >> 4;
       right := (node.siblings & 0x0f);

       IF get_next_bit() THEN BEGIN
           IF right == 0 THEN /* Literal token */
               literal := ByteToChar(get_next_byte());

               RETURN (literal, True);
           ELSE
               RETURN get_next_char(nodelist, index + right)
        END ELSE
                RETURN get_next_char(nodelist, index + left)
    END
END
```

The function get_next_char() is executed until its second return value is True (i.e. if a value was read directly from the input stream) while the first return value equals a certain terminator character, which is the first byte stored in the compressed resource:

Offset	Name	Meaning
0	terminator	Terminator character
1	nodes	Number of nodes
2 + i*2	nodelist[i].value	Value of node #i ($0 \leq$; i $\leq$ nodes)
3 + i*2	nodelist[i].siblings	Sibling nodes of node #i
2 + nodes*2	data[]	The actual compressed data

where nodelist[0] is the root node.

2.3.3 Decompression algorithm COMP3

WriteMe.

2.3.4 Decompression algorithm DCL-EXPLODE

originally by Petr Vyhnak

This algorithm matches one or more of the UNKNOWN algorithms.

This algorithm is based on the Deflate algorithm described in the Internet RFC 1951 (see also RFC 1950 for related material).

The algorithm is quite similar to the explode algorithm (ZIP method #6 - implode) but there are differences.

```
        /* The first 2 bytes are parameters */

P1 = ReadByte(); /* 0 or 1 */
        /* I think this means 0=binary and 1=ascii file, but in RESOURCEs I saw always

P2 = ReadByte();
        /* must be 4,5 or 6 and it is a parameter for the decompression algorithm */


/* Now, a bit stream follows, which is decoded as described below: */


LOOP:
     read 1 bit (take bits from the lowest value (LSB) to the MSB i.e. bit 0, bit 1 et
         - if the bit is 0 read 8 bits and write it to the output as it is.
         - if the bit is 1 we have here a length/distance pair:
                 - decode a number with Hufmman Tree #1; variable bit length, result
                   if L1 <= 7:
                         LENGTH = L1 + 2
                   if L1 > 7
                         read more (L1-7) bits -> L2
                         LENGTH = L2 + M[L1-7] + 2

                 - decode another number with Hufmann Tree #2 giving result 0x00..0x3
                   if LENGTH == 2
                         D1 = D1 << 2
                         read 2 bits -> D2
                   else
                         D1 = D1 << P2  // the parameter 2
                         read P2 bits -> D2

                   DISTANCE = (D1 | D2) + 1

                 - now copy LENGTH bytes from (output_ptr-DISTANCE) to output_ptr
```

```
END LOOP
```

The algorithm terminates as soon as it runs out of bits. The data structures used are as follows:

2.3.4.1 M

M is a constant array defined as M[0] = 7, M[n+1] = M[n]+ 2^n. That means M[1] = 8, M[2] = 0x0A, M[3] = 0x0E, M[4] = 0x16, M[5] = 0x26, etc.

2.3.4.2 Huffman Tree #1

The first huffman tree (Section 2.3.2) contains the length values. It is described by the following table:

value (hex)	code (binary)
0	101
1	11
2	100
3	011
4	0101
5	0100
6	0011
7	0010 1
8	0010 0
9	0001 1
a	0001 0
b	0000 11
c	0000 10
d	0000 01
e	0000 001
f	0000 000

where bits should be read from the left to the right.

2.3.4.3 Huffman Tree #2

The second huffman code tree contains the distance values. It can be built from the following table:

value (hex)	code (binary)
00	11
01	1011
02	1010
03	1001 1
04	1001 0
05	1000 1
06	1000 0
07	0111 11
08	0111 10
09	0111 01
0a	0111 00
0b	0110 11
0c	0110 10
0d	0110 01
0e	0110 00
0f	0101 11
10	0101 10
11	0101 01
12	0101 00
13	0100 11

14	0100 10
15	0100 01
16	0100 001
17	0100 000
18	0011 111
19	0011 110
1a	0011 101
1b	0011 100
1c	0011 011
1d	0011 010
1e	0011 001
1f	0011 000
20	0010 111
21	0010 110
22	0010 101
23	0010 100
24	0010 011
25	0010 010
26	0010 001
27	0010 000
28	0001 111
29	0001 110
2a	0001 101
2b	0001 100
2c	0001 011
2d	0001 010
2e	0001 001
2f	0001 000
30	0000 1111
31	0000 1110
32	0000 1101
33	0000 1100
34	0000 1011
35	0000 1010
36	0000 1001
37	0000 1000
38	0000 0111
39	0000 0110
3a	0000 0101
3b	0000 0100
3c	0000 0011
3d	0000 0010
3e	0000 0001
3f	0000 0000

where bits should be read from the left to the right.

2.3.4.4 Huffman Tree #3

This tree describes literal values for ASCII mode, which adds another compression step to the algorithm.

value (hex)	code (binary)
00	0000 1001 001
01	0000 0111 1111
02	0000 0111 1110

03	0000 0111 1101
04	0000 0111 1100
05	0000 0111 1011
06	0000 0111 1010
07	0000 0111 1001
08	0000 0111 1000
09	0001 1101
0a	0100 011
0b	0000 0111 0111
0c	0000 0111 0110
0d	0100 010
0e	0000 0111 0101
0f	0000 0111 0100
10	0000 0111 0011
11	0000 0111 0010
12	0000 0111 0001
13	0000 0111 0000
14	0000 0110 1111
15	0000 0110 1110
16	0000 0110 1101
17	0000 0110 1100
18	0000 0110 1011
19	0000 0110 1010
1a	0000 0010 0100 1
1b	0000 0110 1001
1c	0000 0110 1000
1d	0000 0110 0111
1e	0000 0110 0110
1f	0000 0110 0101
20	1111
21	0000 1010 01
22	0001 1100
23	0000 0110 0100
24	0000 1010 00
25	0000 0110 0011
26	0000 1001 11
27	0001 1011
28	0100 001
29	0100 000
2a	0001 1010
2b	0000 1101 1
2c	0011 111
2d	1001 01
2e	0011 110
2f	0001 1001
30	0011 101
31	1001 00
32	0011 100
33	0011 011
34	0011 010
35	0011 001
36	0001 1000
37	0011 000
38	0010 111
39	0001 0111
3a	0001 0110

3b	0000 0110 0010
3c	0000 1001 000
3d	0010 110
3e	0000 1101 0
3f	0000 1000 111
40	0000 0110 0001
41	1000 11
42	0010 101
43	1000 10
44	1000 01
45	1110 1
46	0010 100
47	0001 0101
48	0001 0100
49	1000 00
4a	0000 1000 110
4b	0000 1100 1
4c	0111 11
4d	0010 011
4e	0111 10
4f	0111 01
50	0010 010
51	0000 1000 101
52	0111 00
53	0110 11
54	0110 10
55	0010 001
56	0000 1100 0
57	0001 0011
58	0000 1011 1
59	0000 1011 0
5a	0000 1000 100
5b	0001 0010
5c	0000 1000 011
5d	0000 1010 1
5e	0000 0110 0000
5f	0001 0001
60	0000 0101 1111
61	1110 0
62	0110 01
63	0110 00
64	0101 11
65	1101 1
66	0101 10
67	0101 01
68	0101 00
69	1101 0
6a	0000 1000 010
6b	0010 000
6c	1100 1
6d	0100 11
6e	1100 0
6f	1011 1
70	0100 10
71	0000 1001 10
72	1011 0

73	1010 1
74	1010 0
75	1001 1
76	0001 0000
77	0001 111
78	0000 1111
79	0000 1110
7a	0000 1001 01
7b	0000 1000 001
7c	0000 1000 000
7d	0000 0101 1110
7e	0000 0101 1101
7f	0000 0101 1100
80	0000 0010 0100 0
81	0000 0010 0011 1
82	0000 0010 0011 0
83	0000 0010 0010 1
84	0000 0010 0010 0
85	0000 0010 0001 1
86	0000 0010 0001 0
87	0000 0010 0000 1
88	0000 0010 0000 0
89	0000 0001 1111 1
8a	0000 0001 1111 0
8b	0000 0001 1110 1
8c	0000 0001 1110 0
8d	0000 0001 1101 1
8e	0000 0001 1101 0
8f	0000 0001 1100 1
90	0000 0001 1100 0
91	0000 0001 1011 1
92	0000 0001 1011 0
93	0000 0001 1010 1
94	0000 0001 1010 0
95	0000 0001 1001 1
96	0000 0001 1001 0
97	0000 0001 1000 1
98	0000 0001 1000 0
99	0000 0001 0111 1
9a	0000 0001 0111 0
9b	0000 0001 0110 1
9c	0000 0001 0110 0
9d	0000 0001 0101 1
9e	0000 0001 0101 0
9f	0000 0001 0100 1
a0	0000 0001 0100 0
a1	0000 0001 0011 1
a2	0000 0001 0011 0
a3	0000 0001 0010 1
a4	0000 0001 0010 0
a5	0000 0001 0001 1
a6	0000 0001 0001 0
a7	0000 0001 0000 1
a8	0000 0001 0000 0
a9	0000 0000 1111 1
aa	0000 0000 1111 0

ab	0000 0000 1110 1
ac	0000 0000 1110 0
ad	0000 0000 1101 1
ae	0000 0000 1101 0
af	0000 0000 1100 1
b0	0000 0101 1011
b1	0000 0101 1010
b2	0000 0101 1001
b3	0000 0101 1000
b4	0000 0101 0111
b5	0000 0101 0110
b6	0000 0101 0101
b7	0000 0101 0100
b8	0000 0101 0011
b9	0000 0101 0010
ba	0000 0101 0001
bb	0000 0101 0000
bc	0000 0100 1111
bd	0000 0100 1110
be	0000 0100 1101
bf	0000 0100 1100
c0	0000 0100 1011
c1	0000 0100 1010
c2	0000 0100 1001
c3	0000 0100 1000
c4	0000 0100 0111
c5	0000 0100 0110
c6	0000 0100 0101
c7	0000 0100 0100
c8	0000 0100 0011
c9	0000 0100 0010
ca	0000 0100 0001
cb	0000 0100 0000
cc	0000 0011 1111
cd	0000 0011 1110
ce	0000 0011 1101
cf	0000 0011 1100
d0	0000 0011 1011
d1	0000 0011 1010
d2	0000 0011 1001
d3	0000 0011 1000
d4	0000 0011 0111
d5	0000 0011 0110
d6	0000 0011 0101
d7	0000 0011 0100
d8	0000 0011 0011
d9	0000 0011 0010
da	0000 0011 0001
db	0000 0011 0000
dc	0000 0010 1111
dd	0000 0010 1110
de	0000 0010 1101
df	0000 0010 1100
e0	0000 0000 1100 0
e1	0000 0010 1011
e2	0000 0000 1011 1

e3	0000 0000 1011 0
e4	0000 0000 1010 1
e5	0000 0010 1010
e6	0000 0000 1010 0
e7	0000 0000 1001 1
e8	0000 0000 1001 0
e9	0000 0010 1001
ea	0000 0000 1000 1
eb	0000 0000 1000 0
ec	0000 0000 0111 1
ed	0000 0000 0111 0
ee	0000 0010 1000
ef	0000 0000 0110 1
f0	0000 0000 0110 0
f1	0000 0000 0101 1
f2	0000 0010 0111
f3	0000 0010 0110
f4	0000 0010 0101
f5	0000 0000 0101 0
f6	0000 0000 0100 1
f7	0000 0000 0100 0
f8	0000 0000 0011 1
f9	0000 0000 0011 0
fa	0000 0000 0010 1
fb	0000 0000 0010 0
fc	0000 0000 0001 1
fd	0000 0000 0001 0
fe	0000 0000 0000 1
ff	0000 0000 0000 0

where bits should be read from the left to the right.

2.3.5 Decompression algorithm UNKNOWN

The algorithms listed as UNKNOWN-x have not yet been mapped to actual algorithms but are known to be used by the games. For some of them, it is possible that they match one of the algorithms described above, but have not yet been added to FreeSCI in an appropriate way (refer to DCL-EXPLODE for a good example).

Chapter 3

The Graphics subsystem

3.1 General stuff

The graphics in SCI are generated using four resource types:

- Pic resources for background pictures
- View resources for images
- Font resources for drawing text
- Cursor resources for displaying the mouse pointer

Those resources are drawn on three distinct maps:

- The visual map, used for displaying the actual pictures the player sees
- The priority map, which keeps information about how the depth of the screen
- The control map, which contains special information

3.2 SCI Ports

Lars Skovlund

Version 1.0, 6. July 1999

Note that the observations made in this document are generally based on SCI version 0.000.572 (the one that comes with LSL3), but should be valid even for SCI01 and SCI1, as well. I know already about some differences in the port system from SCI0 to SCI1, but I feel we should have an interpreter running for SCI0 before dealing with SCI1.

This article discusses a key data structure in SCI graphics handling; this data structure is called a port, and it is involved in most graphics-related operations. The port is basically a graphics state record, storing things like pen color, current font, cursor position etc. Each port also has an origin and a size. The actual port data structure has remained absolutely unchanged from SCI0 up to the latest versions of SCI1.

The port can be viewed as a rectangle in which things are drawn. Every drawing operation (even KDrawPic) is executed relative to the origin coordinates of the current port (depending on the kernel function, other parameters in the port structure are used as well), such that coordinate (0, 0) in the "picture window" (such a thing really exists in SCI!) is *not* the top of the screen, but rather the leftmost point underneath the menu bar. The coordinate set (0,0) is called the local coordinates, and its physical position on the screen, (0, 10), is called the global coordinates. Kernel calls exist to ease conversion between the two coordinate systems, but they are, it appears, meant for event handlers to use, and not generally usable (I think they take a pointer to an Event object as a parameter).

At least three ports are created and managed automatically by the SCI interpreter. These are the "window manager" port, the menu port, and the picture port (which is actually a window, see later). The latter two should be fairly easy to understand. The menu bar is drawn in the menu port, and the current room is drawn in the picture port. What may be less obvious is that the window manager port

is an "invisible" port, on which the window backgrounds are drawn, although the windows have a port themselves. If you are familiar with Windows™ programming, the term "client rectangle" may ring a bell here - SCI draws the window backgrounds, using values in the window manager port, while the window's own port controls what is drawn inside it. The window manager port covers the same bounding rectangle as the picture window, but it is transparent so it doesn't mess up the graphics.

I feel compelled to mention windows for a bit here, not in depth - they are the subject of a later article - but just to mention that the structure used to manage windows is just an extension of the port structure. Whenever an SCI system call needs a pointer to a port structure, a pointer to a window structure will do. This implicates that the SysWindow class (which implements windows) has no "port" property. Instead, its "window" property points to the extended port/window structure which can safely be passed to KSetPort. Not surprisingly, many of KNewWindow's arguments end up in the port part of the window structure.

An SCI program can't directly instantiate a port. If a program wants to access a specific part of the screen using ports, it has to instantiate a transparent window. In fact, SCI creates the picture window using RNewWindow, the same function that the kernel call KNewWindow ends up calling, asking for an untitled window with a transparent background - but more on that in a later article.

It must be stressed that ports are purely internal structures. Although a program can select different ports to draw in, the data structures themselves are absolutely off-limits to SCI code. KNewWindow fills a port structure with user-supplied data, but there is no way of changing that data, short of disposing the window and instantiating it again. The structure is frequently changed by SCI itself, though.

Only two kernel calls deal directly with ports:

KGetPort (see Section 5.5.2.21)

KSetPort (see Section 5.5.2.22)

These two functions are often used in pairs (also internally), like:

```
var temp;

temp=KGetPort();  /* Save the old port */
KSetPort(...);      /* Activate some other port */
..                            /* Draw some stuff */
KSetPort(temp);    /* Reactivate the old port */
```

3.3 The Cursor resource

This resource stores a simple bitmap describing the shape and texture of the mouse pointer. All information stored herein is little endian in byte order.

0x00 - 0x01 X coordinate of the mouse cursor hot spot as a 16 bit integer. This variable is not used in SCI0.

0x02 - 0x03 Y coordinate of the mouse cursor hot spot as a 16 bit integer. Only 0x03 is used in SCI0; here, if set, the hot spot is at (8,8), if not set, it is located at (0,0).

0x04 - 0x23 This is a list of 16 unsigned 16 bit integers constituting bitmasks for the mouse cursor's transparency map, with the MSB representing the leftmost pixel.

0x24 - 0x43 This is another list of 16 unsigned 16 bit integers. Each of them represents another bitmask, determining whether the mouse cursor pixel should be drawn in black (not set) or white (set).

To determine whether or not to draw a pixel, and, if it is to be drawn, in which color it should be drawn in, the corresponding bits of both bitmask lists mentioned above have to be examined. In the table below, A represents a bit from the first list, and B the corresponding bit from the lower list.

3.3.1 Color mapping for the SCI0 mouse pointer

AB	Result
00	Transparent
01	Transparent
10	0x00 (Black)
11	0x0f (White)

3.3.2 Color mapping for the SCI1 mouse pointer

Since this method of doing things wastes one combination, the table was changed for SCI01 and SCI1:

AB	Result
00	Transparent
01	0x0f (White)
10	0x00 (Black)
11	0x07 (Light Gray)

3.4 The SCI0 View Resource

In SCI0, Views are collections of images or sprites. Each View resource contains a number of groups, which, in turn, contain one or more images. Usually, those groups contain a number of consecutive animation frames. It appears to be customary to store related animations or images in a single frame. For example, the basic movements of all protagonists (four or eight animation cycles (depending on the game)) are stored inside of a single View resource. Please note that the byte order of the following data is always little endian.

3.4.1 The View Resource

0x00 - 0x01 The number of image groups available.

0x02 - 0x03 A bitmask containing the 'mirrored' flag for each of the groups, with the LSB containing the 'mirrored' flag for group 0.

0x04 - 0x07 - unknown -

0x08... A list of indices pointing to the start of the cell list for each image group. The number of entries is equal to the number of cells as described in 0x00 - 0x01.

3.4.2 Cell List

0x00 - 0x01 The number of image cells available for this group.

0x02 - 0x03 - unknown -

0x04... A list of 16 bit pointers indexing the start of the image cell structure for each image cell. The pointers are relative to the beginning of the resource data.

3.4.3 Image Cell

0x00 - 0x01 The horizontal (X) size of the image.

0x02 - 0x03 The vertical (Y) size of the image.

0x04 The x placement modifier. This signed value determines the number of pixels a view cell is moved to the right before it is drawn.

0x05 The y placement modifier. This signed value determines the number of pixels a view cell is moved downwards before it is drawn.

0x06 The color key, i.e. the color number used for transparency in this cell.

0x07... A list of combined color/repeat count entries. Each byte contains a color entry (low nibble) and a repeat count (high nibble). If the color is equal to the color key from index 0x06, then no drawing should be performed, although [repeat] pixels still need to be skipped. It is not known whether this list is terminated; the FreeSCI drawing algorithm stops drawing as soon as the rectangle defined in the first two cell entries has been filled.

3.5 The SCI font resource

SCI font resources remained unchanged during the SCI revisions and were still used in SCI32. Their format is relatively straightforward and completely sufficient for any 8 or even 16 bit character table:

Table 3.1 The SCI font resource data structure

Offset	Type	Meaning
0	16 bit integer, little endian encoding	Always zero (?)
2	16 bit integer, little endian encoding	NUMCHAR: Number of characters
4	16 bit integer, little endian encoding	HEIGHT: Number of pixel lines per text line
6 + NR * 2	16 bit integer, little endian encoding	Absolute offset of the character #NR, where 0 <= NR < NUMCHAR

HEIGHT does not affect the height of a character, though- it only tells the interpreter how far to move downwards when displaying a line of text. The characters referenced to starting at offset 6 are encoded as follows:

Table 3.2 The SCI font resource character data structure

Offset	Type	Meaning
0	unsigned 8 bit integer	character HEIGHT
1	unsigned 8 bit integer	character WIDTH
2...	bitmask, size HEIGHT * round_up(WIDTH / 8)	Bitmask for the character

The bitmap consists of HEIGHT lines of n bytes, where n equals the number of bytes required for storing WIDTH bits. Data is stored with the MSB first, in little-endian encoding (first byte describes the 8 leftmost pixels), where a pixel is drawn iff the bit it corresponds to is set.

3.6 The SCI0 and SCI01 pic resource

The pic (background picture) resource format used in SCI0 is rather complex in comparison to the other graphical resource formats. It is best described as a sequence of drawing operations on a set of four 320x200 canvases, three of which are later used in the game (visual, priority, and control), and one of which is used during the drawing process for auxiliary purposes[1]

In order to describe the process, we will first need to define a set of operations we base them on:

```
FUNCTION peek_input(): Byte; /* returns the byte pointed to by the input pointer */
FUNCTION get_input(): Byte; /* works like peek_input(), but also increminates the
                            ** input pointer  */
FUNCTION skip_input(x): Byte; /* skips x input bytes */
```

Using these pre-defined functions, we will now define additional helper functions used for reading specifically encoded data tuples:

```
FUNCTION GetAbsCoordinates(): (Integer, Integer)
VAR
        x, y, coordinate_prefix : Integer;
BEGIN
        coordinate_prefix := get_input();
        x := get_input();
        y := get_input();
        x |= (coordinate_prefix & 0xf0) << 4;
        y |= (coordinate_prefix & 0x0f) << 8;

        RETURN (x,y)
END


FUNCTION GetRelCoordinates(x : Integer, y: Integer): (Integer, Integer)
VAR
        input : Integer;
BEGIN
        input := get_input();
        IF (input & 0x80) THEN
                x -= (input >> 4);
        ELSE
                x += (input >> 4);
        FI

        IF (input & 0x08) THEN
                y -= (input & 0x7);
        ELSE
                y += (input & 0x7);
        FI

        RETURN (x,y)
END
```

We also need some data types based on EGACOLOR and PRIORITY, which can be thought of as integers:

```
TYPE Palette = ARRAY[0..39] of EGACOLOR[0..1]
```

[1] Due to the vector graphics nature of these drawing operations, they are inherently more scaleable than pixmaps.

```
TYPE Priority_Table = ARRAY[0..39] of PRIORITY

Palette default_palette =
     <(0,0), (1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7),
      (8,8), (9,9), (a,a), (b,b), (c,c), (d,d), (e,e), (8,8),
      (8,8), (0,1), (0,2), (0,3), (0,4), (0,5), (0,6), (8,8),
      (8,8), (f,9), (f,a), (f,b), (f,c), (f,d), (f,e), (f,f),
      (0,8), (9,1), (2,a), (3,b), (4,c), (5,d), (6,e), (8,8)>;

#define DRAW_ENABLE_VISUAL   1
#define DRAW_ENABLE_PRIORITY 2
#define DRAW_ENABLE_CONTROL  4

#define PATTERN_FLAG_RECTANGLE 0x10
#define PATTERN_FLAG_USE_PATTERN 0x20
```

And now for the actual algorithm:

```
FUNCTION DrawPic (cumulative, fill_in_black : Boolean; default_palette: Integer; visua
VAR
        palette : Array [0..3] of Palette;
        drawenable, priority, col1, col2, pattern_nr, pattern_code : Integer;
BEGIN
        palette := (default_palette * 4);
        drawenable := DRAW_ENABLE_VISUAL | DRAW_ENABLE_PRIORITY
        priority := 0;
        col1 := col2 := 0;
        pattern_nr := 0;
        pattern_code := 0;

        IF (!cumulative) THEN BEGIN
                visual_map := (0xf * 320 * 200);
                map control := map priority := map aux := (0 * 320 * 200);
        END

        FOREVER DO BEGIN

                opcode := get_input();

                COND opcode:
                        0xf0 => /* PIC_OP_SET_COLOR */
                                code := get_input();
                                (col1, col2) := palette[default_palette + (code / 40)]
                                drawenable |= DRAW_ENABLE_VISUAL;

                        0xf1 => /* PIC_OP_DISABLE_VISUAL */
                                drawenable &= ~DRAW_ENABLE_VISUAL;

                        0xf2 => /* PIC_OP_SET_PRIORITY */
                                code := get_input();
                                priority := code & 0xf;
                                drawenable |= DRAW_ENABLE_PRIORITY;

                        0xf3 => /* PIC_OP_DISABLE_PRIORITY */
                                drawenable &= ~DRAW_ENABLE_PRIORITY;

                        0xf4 => /* PIC_OP_RELATIVE_PATTERNS */
                                IF (pattern_code & PATTERN_FLAG_USE_PATTERN) THEN
```

```
                        pattern_nr := (get_input() >> 1) & 0x7f
                FI

                (x,y) := GetAbsCoordinates();

                DrawPattern(x, y, col1, col2, priority, control, drawe
                                pattern_code & PATTERN_FLAG_USE_PATTE
                                pattern_size, pattern_nr, pattern_coo

                WHILE (peek_input() < 0xf0) DO BEGIN
                        IF (pattern_code & PATTERN_FLAG_USE_PATTERN) T
                                pattern_nr := (get_input() >> 1) & 0x7
                        FI
                        (x,y) =  GetRelCoordinates(x,y);
                        DrawPattern(x, y, col1, col2, priority, contro
                                        pattern_code & PATTERN_FLAG_U
                                        pattern_size, pattern_nr, pat
                END

        0xf5 => /* PIC_OP_RELATIVE_MEDIUM_LINES */
                (oldx, oldy) := GetAbsCoordinates();
                WHILE (peek_input() < 0xf0) DO BEGIN
                        temp := get_input();
                        IF (temp & 0x80) THEN
                                y := oldy - (temp & 0x7f)
                        ELSE
                                y := oldy + temp
                        FI
                        x = oldx + get_input();
                         DitherLine(oldx, oldy, x, y, col1, col2, prio
                        (oldx, oldy) := (x, y);
                END

        0xf6 => /* PIC_OP_RELATIVE_LONG_LINES */
                (oldx, oldy) :=  GetAbsCoordinates()
                WHILE (peek_input() < 0xf0) DO BEGIN
                        (x, y) := GetAbsCoordinates();
                        DitherLine(oldx, oldy, x, y, col1, col2, prior
                        (oldx, oldy) := (x, y);
                END

        0xf7 => /* PIC_OP_RELATIVE_SHORT_LINES */
                (oldx, oldy) =  GetAbsCoordinates()
                WHILE (peek_input() < 0xf0) DO BEGIN
                        (x, y) := GetRelCoordinates(oldx, oldy);
                        DitherLine(oldx, oldy, x, y, col1, col2, prior
                        (oldx, oldy) := (x, y);
                END

        0xf8 => /* PIC_OP_FILL */
                IF (fill_in_black) THEN
                        (oldc1, oldc2) := (c1, c2);
                FI

                WHILE (peek_unput() < 0xf0) DO BEGIN
                        (x, y) := GetAbsCoordinates();
                        DitherFill(x, y, col1, col2, priority, special
                END
```

```
                IF (fill_in_black) THEN
                        (c1, c2) := (oldc1, oldc2);
                FI

        0xf9 => /* PIC_OP_SET_PATTERN */
                pattern_code := get_input() & 0x37;
                pattern_size := pattern_code & 0x7;

        0xfa => /* PIC_OP_ABSOLUTE_PATTERNS */
                WHILE (peek_input() < 0xf0) DO
                        IF (pattern_code & PATTERN_FLAG_USE_PATTERN)
                                pattern_nr := (get_input() >> 1) & 0x
                        FI
                        (x, y) := GetAbsCoordinates();
                        DrawPattern(x, y, col1, col2, priority, contro
                                        pattern_code & PATTERN_FLAG_U
                                        pattern_size, pattern_nr, pat
                        END

        0xfb => /* PIC_OP_SET_CONTROL */
                control := get_input() & 0x0f;
                drawenable |= DRAW_ENABLE_CONTROL;

        0xfc => /* PIC_OP_DISABLE_CONTROL */
                drawenable &= ~DRAW_ENABLE_CONTROL;

        0xfd => /* PIC_OP_RELATIVE_MEDIUM_PATTERNS */
                IF (pattern_code & PATTERN_FLAG_USE_PATTERN) THEN
                        pattern_nr := (get_input() >> 1) & 0x7f;
                FI

                (oldx, oldy) := GetAbsCoordinates();

                DrawPattern(x, y, col1, col2, priority, control, drawe
                                pattern_code & PATTERN_FLAG_USE_PATTE
                                pattern_size, pattern_nr, pattern_coo

                WHILE (peek_input() < 0xf0) DO BEGIN
                        IF (pattern_code & PATTERN_FLAG_USE_PATTERN) T
                                pattern_nr := (get_input() >> 1) & 0x
                        FI

                        temp := get_input();
                        IF (temp & 0x80)
                                y := oldy - (temp & 0x7f)
                        ELSE
                                y := oldy + temp
                        FI
                        x := oldx + get_input();
                        DrawPattern(x, y, col1, col2, priority, contro
                                        pattern_code & PATTERN_FLAG_U
                                        pattern_size, pattern_nr, pat
                END

        0xfd => /* PIC_OP_OPX */
                COND get_input():
                        0x00 => /* PIC_OPX_SET_PALETTE_ENTRY */
                                WHILE peek_input() < 0xf0 DO BEGIN
                                        index := get_input();
```

```
                                                        color := get_input();
                                                        palette[index / 40][color % 40
                                                END

                                        0x01 => /* PIC_OPX_SET_PALETTE */
                                                palette_number := get_input();
                                                FOR i := 0 TO 39 DO
                                                        palette[palette_number][i] :=
                                                OD

                                        0x02 => /* PIC_OPX_MONO0 */
                                                skip_input(41);

                                        0x03 => /* PIC_OPX_MONO1 */
                                                skip_input(1);

                                        0x04 => /* PIC_OPX_MONO2 */
                                        0x05 => /* PIC_OPX_MONO3 */
                                                skip_input(1);

                                        0x06 => /* PIC_OPX_MONO4 */
                                        0x07 => /* PIC_OPX_EMBEDDED_VIEW */ /* SCI01 o
                                        0x08 => /* PIC_OPX_SET_PRIORITY_TABLE */ /* SC

                        0xff => return (visual, control, priority, aux);
                END OF COND
        END
END
```

This algorithm uses three auxiliary algorithms, DrawPattern, DitherLine, and DitherFill, which are sketched below. All of these functions are supposed to take the four maps as implicit parameters.

```
PROCEDURE DrawPattern(x, y, col1, col2, priority, control, drawenable : Integer;  soli

Alters (x,y) so that 0 <= (x - pattern_size), 319 >= (x + pattern_size), 189 >= (y + p
0 <= (y - pattern_size), then draws a rectangle or a circle filled with col1, col2, pr
as determined by drawenable.
If rectangle is not set, it will draw a rectangle, otherwise a circle of size pattern_
pattern_nr is used to specify the start index in the random bit table (256 random bits



PROCEDURE DitherLine(x, y, xend, yend, color1, color2, priority, control, drawenable

Draws a dithered line between (x, y+10) and (xend, yend+10). If the appropriate drawen
are set, it draws 'priority' to the priority map, 'control' to the control map, and 'c
(alternating) to the visual map. The auxiliary map is bitwise-or'd with the drawenable
done.



PROCEDURE DitherFill(x, y, col0, col1, priority, control, drawenable : Integer)
Fills all layers for which drawenable is set with the appropriate content.
Diagonal filling is not allowed.
Boundaries are determined as follows:
x<0, x>319, y<10, y>199 are hard boundaries. We now determine the
'boundary map' bound_map and the allowed color legal_color.
If bound_map[coordinates] = legal_color, then the pixel may be filled.
```

```
IF (drawenable & DRAW_ENABLE_VISUAL)
        bound_map = visual;
        legal_color = 0xf;
ELSIF (drawenable & DRAW_ENABLE_PRIORITY)
        bound_map = priority;
        legal_color = 0;
ELSIF (drawenable & DRAW_ENABLE_CONTROL)
        bound_map = control;
        legal_color = 0;
ELSE
        return;
FI
```

3.7 SCI1 palettes

3.8 Palette types

There are two kinds of palettes in SCI: Local palettes and global palettes. Local palettes are associated with a graphical resource, while the global palette resides in a separate resource. In SCI1.0, both kinds hold exactly 256 elements, and only synchronous palette operations can be initiated by the VM. SCI1.1 changes the palette format radically and introduces the ability to perform asynchronous palette cross-fades. The exact format of SCI1.1 palettes is not known and will not be described here, nor will the associated kernel calls.
The global palette is updated on several occasions:

- On game startup, the global palette is loaded from the 999.pal resource.
- It may be replaced at any time using the appropriate kernel call.
- When a graphical resource is loaded for display, its palette entries are merged into the global palette ("installing" it – see section 3.10 for more information), and all further operations are carried out on the global palette. Functions that only return view metadata do not touch the global palette.
- The game may explicitly request installation of a view's palette (SCI1.1 only)

The local palette entries are usually placed in the right spot in the local palette, such that installing them is a simple matter of copying. This is not always the case, however.

3.9 The palette format

A SCI1.0 palette, whether global or local, consists of the following items:

1. A mapping of each color index into the global palette (the global palette and most, if not all, palettes on disk have the identity function here).
2. A 32-bit time stamp for internal use (it is always zero in the resources).
3. A list of FRGB tuples where F is a flag byte telling if the index is in use and if it is the victim of an approximate mapping (see section 3.10). The flag bits are given in figure 3.9 – the remaining bits were perhaps used during development.

In addition to these, the global palette contains, for each color, a brightness value. It is measured in percent, and is kept separate from the RGB triples to avoid interfering with color matching. This is not stored in the resources, but defaults to 100. The values may be changed by the game, for example to allow the same pic to be used in day and night scenes.

A palette is assumed always to contain pure black and pure white with indices 0 and 255, respectively. This is enforced when performing cross-fades in SCI1.1, and in addition, those entries are ignored by certain operations, brightness control in particular.

3.10 Installing a palette

When installing a palette, there are two different modes. They are shown in table 3.10 and described in detail below. No, the numbers in the table are not reversed, it just happens that the least-used mode has the value 0.

Forced mapping As the name indicates, *target* palette entries are always overwritten by the corresponding *source* entry. If a *source* entry is unused, the corresponding *target* entry is left untouched.

Normal merge When using this method, five steps are taken in an attempt to map each source entry to a target entry. In each case, the *source* palette is updated to indicate the new mapping. If one step fails, the next is executed, and so on:

1. If the i'th *source* entry is not used, skip to the next.
2. If the i'th *target* entry is not used, then the *source* entry is mapped to it.
3. Try to find an exact match in the *target* palette and map to it if one is found.
4. Try to find an unused index in the *target* palette and map to it if one is found.
5. Map to the closest color in the *target* palette, with infinite tolerance. Because infinite tolerance is used, this step will never fail. In addition, flag bit 4 in the *target* palette entry may be set by some SCI versions to indicate an approximate mapping.

Forced mapping is used implicitly almost everywhere within the interpreter. Thus, there is no real need for removing palette entries explicitly, because a large part of the palette is (in practice, though not in theory) replaced in strategic places. When loading palettes explicitly, the game may specify a different mapping strategy.

Brightness values are left untouched during all implicit palette operations. Thus, it is not safe to add new cels to a scene while using brightness adjustment.

Figure 3.1 Flag bits in each palette entry

Bit	Description
0	The entry is used
4	Another color has been inexactly mapped to this one

Figure 3.2 The palette installation modes

Mode	Description
0	Merge into the global palette.
2	Force insertion.

3.11 Kernel calls

3.11.1 Palette syscall

The subfunctions listed here are for version 1.001.029; the subfunction indices are given in figure 3.3. Palette ranges are closed intervals. The functions are described in detail below.

Subfunctions of the Palette syscall:

(Palette PAL_LOAD_PALETTE number mode)

Loads the palette resource given by *number* and makes it the current global palette. The exact semantics depend on the *mode* parameter, see section 3.10.

(Palette PAL_SET_FLAGS first last bit)

For the range of palette entries givem by *first* and *last*, sets the given *bit*(s) in the palette flags (using the binary OR operator).

(Palette PAL_CLEAR_FLAGS first last bit)

For the range of palette entries givem by *first* and *last*, sets the given *bit*(s) in the palette flags (using the binary NAND operator).

(Palette PAL_SET_BRIGHTNESS first last brightness defer)

For the range of palette entries given by *first* and *last*, sets the brightness (measured in percent) to *brightness*. The *defer* flag, if given, tells SCI whether to defer the changes until later. If the parameter is not given, the changes are always committed immediately.

(Palette PAL_CLOSEST_RGB r g b)

Finds the palette entry that matches the given rgb triple most closely. Infinite tolerance is used – new palette entries are never created.

Returns: The index of the matching palette entry

(Palette PAL_CYCLE first last speed . . .)

Cycles the given range(s) of palette entries once. The *speed* parameter can be used to control the cycling as follows: The interpreter remembers each active cycling range, and stores a timestamp for each of them. We only cycle a particular range if at least *speed* game ticks have passed since the last time we did so. The interpreter is responsible for aging the active cycles and eventually getting rid of them.

An arbitrary number of arguments can be given in groups of three. The given cycles are performed sequentially. A negative *speed* indicates reverse cycling (but the function as a speed setting still applies).

Figure 3.3 Subfunction indices of the Palette() kernel call

Subfunction	Index
PAL_LOAD_PALETTE	1
PAL_SET_FLAGS	2
PAL_CLEAR_FLAGS	3
PAL_SET_BRIGHTNESS	4
PAL_CLOSEST_RGB	5
PAL_CYCLE	6
PAL_SAVE_PALETTE	7
PAL_RESTORE_PALETTE	8

(Palette PAL_SAVE_PALETTE)

Allocates memory for a palette and stores a snapshot of the global palette in it (including brightness values). The memory may be released either by using the **PAL_RESTORE_PALETTE** subfunction or the **Memory** kernel call.

Returns: A pointer to the allocated memory block

(Palette PAL_RESTORE_PALETTE handle)

Restores the contents of a palette handle and implicitly frees the associated memory.

3.12 Windows, Dialogs and Controls

by Lars Skovlund

Version 1.0, 7. July 1999

I am going to start by mentioning the menus. It has nothing to do with the material I deal with in this essay. They use different kernel calls, and such things as port management are handled internally by the kernel routines. The SCI program just sets up a menu structure using the kernel calls. Since they are irrelevant to the subject of this essay, I will not spend more time on them.

The Rect structure is important (also to ports) since it is the basis for passing a screen position to the interpreter. It looks like this:

```
typedef struct
{
  short top, left, bottom, right;
}
```

It will be seen from this that rectangle coordinates in SCI are not normally represented in the usual (x,y,width,height) fashion. So pay close attention to this structure! Also, it is not passed as a pointer, but rather as the four values in order. This is particularly true of SCI objects, where the property names nsTop etc. actually form a Rect structure which can be used directly by the interpreter.

Windows are created using the KNewWindow kernel function. Each window has six attributes which are passed from the script to the kernel function:

Bounding rectangle
Title
Type
Priority
Foreground color
Background color

Of these, the type and priority are the most interesting, because they decide the appearance of the window. The type is a bit field:

bit 0 - transparency
bit 1 - window does _not_ have a frame
bit 2 - the window has a title
bit 3-6 - unused
bit 7 - see below

Bit 0 specifies a transparent window. KNewWindow does not save the image behind the created window - it stays on the screen until the pic is redrawn, so windows with this style definitely can't be used as message boxes. It does have some special uses, though. If this bit is not set, KNewWindow draws a rectangle in the specified background color using the bounding rectangle coordinates (using the WM port). When this bit is set,

Bit 1 specifies a window without a frame. The frame is the black shading you can see in the corner of a message box.

Bit 2 tells KNewWindow to draw a grey title bar with a title printed in white. In the version I have used for this essay, it is not possible to change the title bar colors. Note that the bounding rectangle is always specified as if the window had no title bar. If this bit is set, ten pixels are reserved above the coordinates specified. Although this bit is set, the Title parameter may still be NULL. If this is the case, an empty title bar is drawn.

Bit 7 has a special meaning; it is used only in window type 0x81, and is not tested in any other way. When this style is chosen, KNewWindow does not draw anything at all. It is the caller's responsibility to draw a window frame on the WM port. CB1 uses this style for its ornate windows, and draws the frame manually.

The picture window which I mentioned in the last article is created using style 3 (that is, TRANSPARENT | NOFRAME). The normal message box styles used in LSL3 are 0 and 4.

I have not been able to investigate the priority property yet, so the fol- lowing is based on suppositions. It is only used when drawing transparent windows. In this case, if priority is not -1 (which means not used), the window is drawn onto the priority map (with the specified priority value) as well as the screen.

There is a class called SysWindow which is just a simple wrapper around the following two kernel calls. Try breaking on SysWindow::open, then type c to inspect the current object. You can change all the parameters to KNewWin- dow (the Rect is split in its fields, to nsTop, nsLeft etc.)

To create a window structure, use KNewWindow (see Section 5.5.2.20); to remove it again, apply KDisposeWindow (see Section 5.5.2.23) on it.

So how do we put stuff inside these windows? That question is a little com- plicated to answer, because it is really a shared effort between the inter- preter and the object hierarchy, and this is one case where the interpreter actually interacts with the objects itself. I will start by explaining the classes involved.

All control types are descendants of a common class (I do not know its name, since it appears to have an invalid name property). Among other things, this common class contains a type number and a state. The type number is the only thing that distinguishes the control types from each other inside the interpreter - if a wrong type is set, the interpreter might try to change a non-existent property.

The type numbers are laid out as follows:

1 - Button control
2 - Text control
3 - Edit control
4 - Icon control
5 - not used
6 - Selector control (as in the Save and Restore boxes)

The gauge "controls" are not really controls. I don't know how they work (yet).

Each control also has a state value. These are laid out as follows:

bit 0	selectable. If this bit is set, the control can be selected using the Tab key. Except for the text and icon controls, all controls are selectable.
bit 1	unknown. Always set, except for the text and icon controls
bit 2	disabled. When this bit is set, a button is grayed out. No other control types are affected.
bit 3	selected. When this bit is set, a frame is drawn around the control.

Note that state 3 is by far the most common. With that explained, I'll move on to the kernel functions. There are three functions associated with controls - KDrawControl (see Section 5.5.2.24), KHiliteControl (see Section 5.5.2.25) and KEditControl (see Section 5.5.2.26). Note that there is a KOnControl kernel call which is entirely unrelated to window management.

The dialogs are implemented using not one, but two classes - Dialog and Window. While the Window class maintains the window (It is derived from SysWindow), the Dialog class is just a list of controls. It is derived from the List class, but has extended functionality to tell its members to redraw etc. There is a special function, located in script 255, which allows scripts to push information about the dialog on the stack instead of creating the Dialog object manually.

Note that the internal debugger uses the same window calls as the SCI script. That is why the screen messes up if you step through drawing code - the debugger has activated the Debug window port, and "forgets" to switch back while stepping across instructions. Thus, all graphics commands are redirected to the debug window port. Not a pretty sight.

3.13 Pictures and movement control

ByLars Skovlund

Version 1.0, 24. July 1999

A pic in SCI consists of three layers (called maps - they are unrelated to the map resources found in SCI1 games). The visual map, used for the picture which appears on the user's screen. The priority map

which tells the interpreter which things go in front of which in the three-dimensional room. Without the priority map, a room would just be a flat, painted surface. The control map decides where game characters (called actors) can walk and where special events occur. These special events are triggered by a game character walking on a particular spot. Where the visual map is almost always very complex and using dithered fills etc., the latter two consist of large areas of solid color.

Many functions which need to access these maps do so by using a bit-field. The bits are laid out as follows (but don't set more than one at a time!)

bit 0 - Visual
bit 1 - Priority
bit 2 - Control

It is important to understand that, although being represented as colors on the screen, a priority/control "color" should be considered a number. The colors map to values according to the standard EGA color values.

Every animated object in SCI has a priority. As the object moves, its pri- ority changes according to the so-called priority bands, explained next (it is, however, possible for a script to lock the priority of a view). The picture window is divided vertically into 16 priority bands. The priority of an animated object is determined by the position of its "base rectangle" in one of these bands. Things are drawn in order of ascending priority, so objects with priority 15 are guaranteed to be in front of everything else. The default priority mapping gives priority 0 a fairly large space, the 42 topmost rows (including the menu bar which AFAIK is 10) in the picture. All other priority bands have the same size. A script can choose to alter this mapping, specifying the amount of space to assign to priority 0, and the number of the last row to include in the mapping calculation.

In most rooms, it is desirable to limit actor movement, confining the actor to a specific part of the screen. In other cases, special events are triggered by movement into a specific screen area. On some occasions, even room switches are implemented using control polygons. While the meaning of priorities is determined by the kernel, the meaning of control values is entirely up to the script. It is more or less a standard, however, that actors can't walk on white control areas.

As the control map is not consulted by the interpreter itself (except in a few cases), scripts need a way to do so. That way is called OnControl, and it is a kernel call. Supplied with a point or a rectangle, it returns a bit mask containing the control values of all the pixels in the desired region. If a specific control value is encountered, it is used as a bit number, and that bit is set in the output mask.

This bit mask system is also used in another place, namely the illegalBits selector of the Act (actor) class. The illegalBits selector determines in which areas the actor may not walk.

The OnControl() system call is explained in Section 5.5.2.96.

Chapter 4

The Sound subsystem

4.1 The SCI0 Sound Resource Format

by Ravi Iyengar
Revision 10, Mar. 11, 2002

4.1.1 Preface

Sierra's SCI0 sound resources contain the music and sound effects played during the game. With the introduction of SCI, the company took advantage of new sound hardware which allowed for far better music than the traditional PC speaker could ever create. Sierra chose two devices to specifically target: the MT-32, and the Adlib. The MT-32 is a MIDI synth while the Adlib is a less expensive card based around the OPL2, a non-MIDI chip. Anyone interested in Sierra music and its history can find information at the Sierra Soundtrack Series (http://www.queststudios.com).

Music is stored as a series of MIDI events, and the sound resource is basically just a MIDI file. The MIDI standard and device implementations are not covered here in detail, but specifications should be readily available elsewhere.

SCI0 Sound resources can also contain digital samples, although an SCI remake of KQ1 is the only DOS game I know of that includes them. These files still contain MIDI data, but the wave data is appended at the end. The MIDI data is an approximation of the sound effect for hardware that can't play digital sound.

Some people prefer the one-based numbering system for channel and program numbers. I personally prefer the zero-based system, and use it here. If you're familiar with channels 1-16, be aware that I will call them 0-15. My intention is not to be deviant from other programs but to be more accurate in representing the way information gets stored. The same is true for programs 0-127 as opposed to 1-128. For whatever reason, convention already holds that controls be numbered 0-127, so nothing in my treatment of them should be abnormal.

Sierra changed its sound file format in the switch to SCI1. I refer only to SCI0 sound files in this specification. Hybrid interpreters such as the one used for Quest for Glory II are also excluded. Finally, SCI games written for non-DOS systems may have different formats. This document applies to Sierra's IBM games.

Please post comments or questions to the SCI webboard:
`http://pub48.bravenet.com/forum/show.asp?usernum=4071584210`

You can contact me personally at ravi.i@softhome.net, but I would prefer that SCI messages be posted on the webboard so everyone can see them.

4.1.2 Sound Devices

A gamer's sound hardware greatly affects how music will sound. Devices used by SCI0 can be broken into general categories:

MIDI Synths These will generally give the best sound quality. MIDI synths are polyphonic with definable instruments through patch files and full support for MIDI controls. The General MIDI standard had not been written when Sierra began writing SCI games, and as far as I know no SCI0

game uses a GM driver or includes a GM track. This means that synths had to be individually supported.

Non-MIDI Synths Generally not as good as MIDI synths, but also less expensive. The OPLx family of chips are still very common among home PC users thanks to the Adlib and SoundBlaster cards. Synths are polyphonic with definable instruments through patch files, but drivers must be written to interpret MIDI events and turn them into commands the hardware will recognize. Support for most sound controls gets lost in the process. Furthermore, drivers must map logical, polyphonic MIDI channels to physical, monophonic hardware channels. A control (4Bh) was introduced for this purpose and will be discussed later.

Beepers Beepers produce very poor music and don't support instrument definitions, but all PC users have one so supporting them covers people without special sound hardware. The most common device is the PC speaker, which is monophonic. Another is the Tandy speaker with 3 channels. Drivers must interpret MIDI events, but need only concern themselves with basic functionality. Interpreting the MIDI events is also made easier because each channel is monophonic. To play a chord on the Tandy, for example, each voice must be put in a separate MIDI channel.

Wave Devices Wave devices play digital sound data. They could be used in conjunction with one of the above devices to add special sound effects to a game. The Amiga port of SCI uses a wave device to play music.

With such a diverse group of devices to support, Sierra put a lot of the work on the shoulders of the drivers. Functions for loading patch files, handling events, pausing, etc. are all in the drivers. The interpreter calls them as needed but does not concern itself at all with how they get implemented.

Listed here are devices supported by the SCI0 interpreter with a little information about each. There could very well be other hardware not listed here, so please send in any missing information.

Device Name	Driver	Patch	Poly	Flag
Roland MT-32	mt32	001	32	01h
Adlib	adl	003	9	04h
PC Speaker	std	*	1	20h
Tandy 1000 or PCJr	jr	*	3	10h
Tandy 1000 SL, TL	tandy	*	3	10h
IBM Music Feature	imf	002	8	+
Yamaha FB-01	fb01	002	8	02h
CMS or Game Blaster	cms	101	12	04h
Casio MT540 or CT460	mt540	004	10	08h
Casio CSM-1		007		
Roland D110,D10,D20		000		
Amiga Sound	amigasnd		4	40h
General MIDI		004		01h

(thanks to Shane T. for providing some of this). Blank fields are unknown, not unused.

* when asked which patch to load, the PC and Tandy speaker drivers return FFFFh, which is a signal that they do not use patches

\+ the imf driver almost certainly uses 02h for the play flag, but I haven't confirmed this

The driver column holds the file name of each driver without the .drv extension. The patch column specifies which patch resource each driver requests. The poly column is the maximum number of voices which can be played at once according to the driver. The flag column gives each device's play flag. Play flags, explained in the header section, determine which channels a device will play.

4.1.3 File Format

Sound files follow the same format as all extracted SCI0 resources. The first two bytes of the file contain a magic number identifying the resource type. The rest of the file contains a dump of the uncompressed data. The identifier is the resource type (04h for sound) OR-ed with 80h and stored as a word. The result will be 84h 00h in extracted sound files.

The sound resource data itself is a header with channel initialization followed by a series of MIDI events.

4.1.3.1 Header

The header provides the sound driver with 2 pieces of information about each channel. The first is a byte which specifies how many voices each logical MIDI channel will be playing. For MIDI synths, this information is not really necessary and is probably ignored. The same goes for beepers. This byte is only useful for non-MIDI synths which must know how many hardware channels each logical MIDI channel will need. This value is only an initial setting. Sound files can request changes to the mapping later with control changes. Requesting more hardware channels than are actually available can cause errors on some drivers.

The second byte describes how the user's sound hardware should treat the channel. It is the combination of bit flags which may be OR-ed together. If the appropriate bit is set for the currently selected sound device, the channel will be played. If it is not, the channel will be silent. The driver decides which bit it will use as the play flag, and the table under Sound Devices lists the flag used by each driver. Drivers ignore the first byte (used to request hardware channels) on MIDI channels they don't play.

The MT-32 always plays channel 9, the MIDI percussion channel, regardless of whether or not the channel is flagged for the device. Other MIDI devices may also do this.

A byte at the beginning of the file, before channel initialization, specifies whether the resource contains a digital sample or not. A value of 0 means that there is only MIDI data. A value of 2 means that there is a digital sample at the end of the file. In this case, only the first 15 MIDI channels have header bytes. The two header bytes for the last channel is replaced with an offset to the digital sound effect. The offset is stored in big-endian order in the resource. If present, it points to the last byte before the digital sample header. If the offset is 0, the file must be searched for the status FCh, and the digital sample header will come next. There may be two FCh bytes in a row, in which case both will come before the digital sample header. The digital sample header is discussed in more detail in the digial sample section.

The header format:

1 byte - digital sample flag (0 or 2)
2 bytes - initialization for channel 0
2 bytes - initialization for channel 1
.
.
.
2 bytes - initialization for channel 15 OR offset to digital sample

The header is always 33 bytes long.

4.1.3.2 Events

The actual music is stored in a series of events. The generic form for an event is:

<delta time> [byte - status] [byte - p1 [p2]]

Delta time is the number of ticks to wait after executing the previous event before executing this event. Ticks occur at 60 Hz. The delta time value is usually a single byte. However, longer delays can be produced by using F8h any number of times before the delta time value. Each F8h byte causes a delay of 240 ticks before continuing playback. For example, the sequence F8 F8 78 FC waits 600 ticks then stops the sequence because of the FCh status. The fact that F8h waits F0h ticks makes me think that E9h is the largest technically allowable delta time.

The delta time must be present in most events. The only exception is when FCh is the status, because FCh is a real-time message. Sierra's resources seem to have always provided a delta time, though. Note also that FCh cannot be used as a delta time value - it will be interpreted as a stop sequence status.

The status byte is basically a command. The most significant bit is always set. This feature is important because the status byte will not always be present. A missing status byte is known as running status mode and the last status gets repreated with the new parameters. Parameters will never have their most significant bits set.

The generic form for a status byte is (in bits) 1xxxcccc. The lower nibble usually specifies a channel. The upper specifies a status.

4.1.3.3 Status Reference

8x n v Note off: Stop playing note n on channel x, releasing the key with velocity v. If a hold pedal is pressed, the note will continue to play after this status is received and end when the pedal is released.

9x n v Note on: Play note n on with velocity v on channel x. Playing a note with velocity 0 is a way of turning the note off.

Ax n p Key pressure (after-touch): Set key pressure to p for note n on channel x. This is to modify key pressure for a note that is already playing.

Bx c s Control: Set control c to s on channel x. This can be confusing because there isn't just one meaning. Changing the settings on different controls will, of course, have different outcomes.

Controls which handle any value are continuous controllers. They have a continuous range. Controls which are only on/off are switches. Their defined values are 01h (OFF) and 7Fh (ON).

Listed in this reference are the non-standard MIDI controls I've found in Sierra SCI0 sound files. Standard controls are not listed here. Not all drivers support all controls.

Control Reference

4Bh Channel mapping: When a channel sets this control, it tells the driver how many notes it will be playing at once, and therefore how many hardware channels it occupies.

4Ch Reset on PauseSound: An on/off switch where a value of zero is off and a non-zero value is on. Note that this is not the same as for standard MIDI control switches. When this control is on, calling the sound driver's PauseSound subfunction will reset the sound position to the beginning. The initial value is set to off when a sound gets loaded.

4Eh Unknown: Experiments in setting and clearing it show that a value of 0 will cause notes to be played without regard for the velocity paramater while a value of 1 will enable velocities.

50h Reverb: I know little about this myself. Rickard Lind reports that it exists in the MT-32 driver and supports parameter values 0-10 (possibly 0-16?).

60h Cumulative cue: The interpreter can get cues from the sound file, which sets the Sound object's signal property. When a sound gets loaded, the initial cue is set to 127. When a CC60 occurs, the new control value is added to the current cue. If the cue were 130, for example, a CC60 5 on any channel would make the new cumulative cue equal 135.

Cx p Program change: Set program (patch / instrument / ect.) to p for channel x. This is a simple instrument change.

Channel 15, however, includes two special cases of this status. The first relates to communication with the game interpreter. If p is less than 127 then the signal property for the game interpreter's Sound object gets set to p, triggering a non-cumulative cue.

If p is equal to 127, then the current position within the sound resource is remembered as the loop point. Normally the driver loops to the beginning of the sound when the sequence ends. If an explicit loop point is set, the sound will be replayed from the marked point instead.

The actual time of the loop point is better explained with a short diagram:

```
0x10 0x91 0x20 0x20  play a note on channel 1
```

```
0x05 0x91 0x20 0x00  stop the previous note
0x00 0x92 0x30 0x10  play a note on channel 2
  [restart here]
0x00 0xCF 0x7F       set loop point
0x00 0xC8 0x05       change to program 5 on channel 8
0x00 0xCF 0x13       set signal to 19
0x20 0xFC            end of file, loop to marked location
```

In both situations (p $<$ 127 and p = 127), no actual program change takes place. Channel 15 is used for control, not playing music.

Dx p Pressure (after-touch): Set key pressure to p on channel x. This is similar to Ax but differs in its scope. Message Ax is applied on a per-note basis while message Dx is applied to an entire channel.

Ex t b Pitch wheel: Set the pitch wheel to tb. The setting is actually a 14 bit number with the least significant 7 bits stored in b and the most significant 7 bits stored in t. The range of values is 0000h to 3FFFh. A value of 2000h means that the pitch wheel is centered. Larger values raise pitch and smaller values lower it.

F0 Begin SysEx: Starts a system exclusive data block. The block must terminate with F7h.

F7 End SysEx: Ends a system exclusive data block. Normal sound data resumes at this point.

FC Stop Sequence: This is a system real-time message which tells the sound driver to stop the current sound. The sound object's signal property gets set to FFFFh and the position moves to the loop point, which defaults to the beginning. Drivers allow this message to occur without a delta time, but I haven't seen any examples.

4.1.4 Digital Samples

The digital sample header is 44 bytes long. Offset 14 in the header contains the frequency as a short integer. Offset 32 contains the sample length, also as a short integer. Other fields in the header are unknown (to me) at the time of writing, but aren't critical to playback.

The wave data comes immediately after the header, stored in unsigned 8 bit PCM format.

4.1.5 Amiga Sound (SCI0)

The SCI0 Amiga Sound driver does not use a patch resource, instead it loads an external instrument bank called 'bank.001'. This file has the following structure (all numbers are big-endian):

`[00]..[07]`	String "X0iUo123"
`[08]..[25]`	Bank name
`[26][27]`	Number of instruments (= #i)
`[28]..`	#i instruments

An instrument has the following format:

[00][01]	Instrument number
[02]..[1f]	Instrument name
[20][21]	Flags: Bit 0 looping on/off Bit 1 pitch changes on/off
[22]	Transpose value in semitones (= #t)
[23][24]	Segment 1 size in words (= #s1)
[25][26][27][28]	Segment 2 offset in bytes
[29][2a]	Segment 2 size in words (= #s2)
[2b][2c][2d][2e]	Segment 3 offset in bytes
[2f][30]	Segment 3 size in words (= #s3)
[31]..[3c]	Velocity envelope
[3d]..	#s1+#s2+#s3 signed 8-bit samples

A velocity envelope has the following format:

[00]	Phase 1 period size in ticks (= #p1)
[01]	Phase 2 period size in ticks (= #p2)
[02]	Phase 3 period size in ticks (= #p3)
[03]	Phase 4 period size in ticks (= #p4)
[04]	Phase 1 velocity delta (= #d1), range [0-64]
[05]	Phase 2 velocity delta (= #d2)
[06]	Phase 3 velocity delta (= #d3)
[07]	Phase 4 velocity delta (= #d4)
[08]	Phase 1 target velocity (= #v1), range [0-64]
[09]	Phase 2 target velocity (= #v2)
[0a]	Phase 3 target velocity (= #v3)
[0b]	Phase 4 target velocity (assumed to be 0) (= #v4)

With looping off, all samples are played. The segments and the envelope data are ignored. With looping on, Segment 1 is played first, followed by a looping of Segment 2 (Segment 3 is never played). As Segments 1 and 2 may overlap; it is possible for #s1 + #s2 to exceed the total number of samples; in that case #s3 will be negative.

Velocity envelope phases 1 and 2 are applied after note-on, and phases 3 and 4 after note-off. If #p0 is zero, no velocity envelope is applied. For other phases, a period size of 0 is interpreted as a period size of 256 ticks. If the velocity drops to 0 at any point, the note is stopped right away, even if more phases follow. Otherwise, the note is stopped after phase 4, which always has a target volume of 0 (note that it is possible to construct velocity envelopes that never terminate).

Each envelope phase n operates as follows, where #v0 is the velocity from the note-on event (divided by two to scale it to Amiga volume levels):

```
vel = #v(n-1);
while (true) {
    set_channel_velocity(vel * #v0 / 64);
    wait_ticks(#pn);
    vel -= #dn;
    if ((#dn >= 0 and vel <= #vn) or (#dn < 0 and vel >= #vn)) {
        if (#vn == 0)
            stop_note();
        break;
    }
}
```

All instruments have a samplerate of 20000Hz. With pitch changes off, the instrument is always played at this frequency, regardless of the note. With pitch changes on, #t is first added to the note. The instrument is then played at the corresponding frequency (where note 101 equals 20000Hz).

The Amiga has four audio channels. Channels 0 and 3 are panned hard left and channels 1 and 2 are panned hard right. The first MIDI channel with playmask 0x40 is mapped to channel 0, the second to channel 1, etc. The driver seems to ignore all pan and volume commands.

4.1.6 General MIDI and MT-32 (SCI1)

The SCI1 MT-32 driver uses patch file 1, and the GM driver uses patch file 4. The file formats are identical however, and the same goes (to a large extent) for the drivers. The patch files have the following structure:

[000]..[07f]	patchMap
[080]..[0ff]	patchKeyShift
[100]..[17f]	patchVolumeAdjust
[180]..[1ff]	percMap
[200]	percVolumeAdjust (not used)
[201]..[280]	velocityMapIndex
[281]..[300]	velocityMap 0
[301]..[380]	velocityMap 1
[381]..[400]	velocityMap 2
[401]..[480]	velocityMap 3
[481][482]	Size (little endian) of MIDI data (= #i)
[483]..	#i bytes of MIDI data

- patchMap[i]: Native patch number for patch i, or -1 for unused entries.
- patchKeyShift[i]: Key shift value for patch i. This value should be added to every non-rhythm key that is played. If the key goes out of bounds [0,127], it should be clipped by a multiple of 12 semitones.
- patchVolumeAdjust[i]: Volume adjust value for patch i. This value should be added to the volume (when setting controller 07h) *before* scaling it by the master volume.
- percMap[i]: Native key for key i of the percussion channel, or -1 for unused entries.
- percVolumeAdjust: Not used.
- velocityMapIndex[i]: Specifies which velocityMap to use for patch i.
- velocityMap[i]: Native velocity for velocity i.
- MIDI data: MIDI data to initialize the device. Note that this data can contain sysex commands, after which an appropriate delay should be executed.

4.1.7 Revision history

Revision 10 - Mar. 11, 2002
- Added section on digital samples (thanks to the FreeSCI developers, Rickard Lind especially)
- Added wave devices to the hardware category list
- Updated header section to cover the header for PCM resources
- Added more play flags to the sound driver table
- Fixed a typo in the sound driver table where I accidentally called the ”Yamaha FB-01” the ”Yamaha FM-01”

Revision 9 - Jul. 4, 2001
- Changed StopSound to PauseSound for control 4Ch
- Updated URL for SCI messageboard
- Added web links for more SCI information[1]
- Did a little proofreading and editing

Revision 8 - Dec. 21, 2000
- Added suggested limit on delta time values
- Fixed hex notation (sometimes listed NNh, sometimes 0xNN)

[1] Editor’s note: These are not included in the FreeSCI documentation version

- Removed notice about early revisions' mistake describing the header's channel mapping byte
- Added note about control 50h (thanks to Rickard Lind)
- Listed MT-32 play flag
- Added notice about the special case of channel 9 to the header section

Revision 7 - Jan. 7, 2000
- Added information about F8h delta times (thanks to Rickard Lind for bringing these to my attention)
- Reorganized Fx status information
- Fixed major error in description of loop points (sorry)
- Fixed typos

Revision 6 - Sep. 17, 1999
- Added information about cues
- Updated control 60h information
- Added information about loop points
- Updated control 4Ch information
- Cleaned up control reference introduction

Revision 5 - Jul. 5, 1999
- Rewrote much of the specification, trying to focus less on explaining MIDI and more on explaining sound resources
- Removed information about standard MIDI controls
- Added driver table
- Expanded sound device section
- Completed header information

Revision 4 - Jun. 19, 1999
- Fixed the list of changes in Revision 3 (was incomplete)
- Expanded the introductory blurb about controls
- I began working with a disassembly of ADL.DRV, and am hoping to use it to complete this specification. The next revision should be more interesting than this one.

Revision 3 - May 4, 1999
- Removed the "compatible games" list. I haven't found a non-compatible SCI0 game yet, which makes the list quite useless.
- Verified that SCI1 sound resources are different.
- Tidied the "About the output medium" section. Does that term "output medium" sound wordy or unclear? I don't really like it, but I didn't want to beat "sound device" to death.
- More information about the header
- Modified the explanation for message FCh.
- Changed most references to status bytes as "commands" with "messges" to stay more consistent with MIDI terminology.
- Added midi.org as a source for more MIDI information
- Removed labels like "tentative" and "incomplete" as things become more concrete – not complete yet, but getting there.
- More information about controls

Revision 2 - Jan. 16, 1999
- Got rid of the HTML. I originally intented to post this as a message on the webboard, but ended up distributing the file. If I'm going to distribute it as a file, there's no need to bother with the HTML since I can do all my formatting as plain text.
- I found refrences to command 8x in the 1988 Christmas Card, so my comment about not seeing one got removed. To date, I haven't seen any examples of commands Ax or Dx.

- Expanded the header section.
- Added information about controls.
- Added information about the output mediums.
- Tried to be more consistent with terminology

Revision 1 - Dec. 29, 1998
- First release of the specification

4.2 Mapping instruments in FreeSCI

4.2.1 The Patch.002 resource

As Ravi describes in his description of the patch resources (which have not been included here), one of the major problems with SCI sound support is the lack of General Midi (GM) support in the earlier games. Since those were written before GM was conceived, this can hardly be considered to be Sierra's fault; but this fact doesn't help when it comes to supporting the games in a portable manner.

Unfortunately, almost every SCI0 game uses an individual instrument mapping scheme. This means that there are only two options to generate GM music from the original SCI sound resources: Either create a manual mapping for each game, or abuse existing data from the game for this purpose. Obviously, the latter way would be either impossible or much easier.

So, the solution would be to use an existing instrument mapping scheme. Those mapping schemes are stored in the patch resources, and, as such, easily accessible to an SCI engine. As those patch files are driver dependant (which, in turn, are hardware dependant), most of the patch data is unusable. The Adlib data, for example, will only work for an OPL-2 FM synthesizer chip or one of its successors, the MT-32 data (which consists of one massive sysex block) won't help anyone without an MT-32 or LAPC-1, and so on. So, to recycle this hardware-dependant data, two new possibilities remain: Either extract and interpret the patch data using a portable software synthesizer (such as timidity), or extract instrument names and map those to GM instruments. The first approach would, of course, yield the better results (at the cost of computation power); but the only software emulator for a specific sound system I've seen so far was an OPL-2 emulator. So the alternative, extracting a text ID of each instrument and using it to map this instrument to a GM instrument, looks much more promising.

Now, most SCI0 games come with a patch.002 resource, which is used by the IBM Music Feature card and Yamaha FM-01 sound synthesizers (both of which appear to use frequency modulation). This is the only patch file that includes text descriptions of most of its instruments. Note this, not all instruments have name representation. This means that some of them can't be mapped and have to be silenced; but those instruments are either used for sound effects only or not used at all, so this isn't critical.

Using those 7-letter instrument names, it is now possible to build a small database of instruments, which, subseqently, can be mapped to GM instruments.

The file structure is relatively simple (for this purpose): Every patch.002 consists of either one or two instrument banks carrying 48 instruments each. Every instrument has a fixed block size of 0x40 bytes; each block starts with the 7-letter description of the instrument or seven blanks if none is available.

If two banks are present, the second bank is separated from the first one by a two-byte sequence (0xab, 0xcd). Keeping this in mind, it is trivial to extract the instrument names of the 48 or 96 instruments.

4.2.2 Percussion instruments

Percussion instruments are treated specially in the MIDI standard. MIDI channel 10 (or 9, if you count from 0 to 15 like most people do) is reserved for percussions and some special effects; each key for this channel represents either nothing or one fixed percussion instrument.

At first glance, this might lead to an additional problem of mapping those percussion instruments. Fortunately, the General Midi standard extends on the MT-32 percussion mappings, which are used in SCI0, so that channel 9 can be left completely untouched in the process of instrument mapping.

Chapter 5

The SCI virtual machine

5.1 Introduction

5.1.1 Script resources

Like any processor, the SCI virtual machine is virtually useless without code to execute. This code is provided by script resources, which constitute the logic behind any SCI game.

In order to operate on the script resource, those first have to be loaded to the heap. The heap is the only memory space that the VM can work on directly (with some restrictions); all other memory spaces have to be used implicitly or explicitly by using kernel calls. The heap also contains a stack, which is heavily used by SCI bytecode.

Each script resource may contain one or several of various script objects, listed here:

Type 1: Object
Type 2: Code
Type 3: Synonym word lists
Type 4: Said specs
Type 5: Strings
Type 6: Class
Type 7: Exports
Type 8: Relocation table
Type 9: Preload text (a flag, rather than a real section)
Type 10: Local variables

Standard SCI0 scripts (of post-0.000.396 SCI0, approximately) consist of a four-byte header, followed by a list of bytes:

`[00][01]`: Block type as LE 16 bit value, or 0 to terminate script resource
`[02][03]`: Block size as LE 16 bit value; includes header size
`[04]...`: Data

The code blocks contain the SCI bytecode that actually gets executed. The export block (of which there may be only one (or none at all)) contains script-relative pointers to exported functions, which can be called by the SCI operations `calle` and `callb`. The local variables block, which stores one of the four variable types, is used to share variables among the objects and classes of one script.

But the most important script members are Objects and Classes. As in the usual OOP terms, Classes refer to object prototypes, and Objects are instantiated Classes. However, unlike most OOP languages, SCI treats the base class very similar to objects, so that they may actually get called by the SCI bytecode. Therefore, they also have their own space for selectors (see below). Also, each object or class knows which class it inherits from and which class it was instantiated from (in the case of objects).

Note that all script segments are optional and 16 bit aligned; they are described in more detail below:

5.1.1.1 Object segments

Objects look like this (LE 16 bit values):

`[00][01]`: Magic number 0x1234
`[02][03]`: Local variable offset (filled in at run-time)
`[04][05]`: Offset of the function selector list, relative to its own position
`[06][07]`: Number of variable selectors (= #vs)
`[08][09]`: The 'species' selector
`[0a][0b]`: The 'superClass' selector
`[0c][0d]`: The '–info–' selector
`[0e][0f]`: The 'name' selector (object/class name)
`[10]...`: (#vs-4) more variable selectors
`[08+ #vs*2][09+ #vs*2]`: Number of function selectors (= #fs)
`[0a+ #vs*2]...`: Selector IDs for the functions
`[08+ #vs*2 + #fs*2][09+ #vs*2 + #fs*2]`zero
`[0a+ #vs*2 + #fs*2]...`: Function selector code pointers

For objects, the selectors are simply values for the selector IDs specified in their species class (which is either present by its offset (in-memory) or class ID (in-script)- the same for the species' superclass (superClass selector)). Info typically has one of the following values (although this does not appear to be relevant for SCI):

`0x0000`: Normal (statical) object
`0x0001`: Clone
`0x8000`: Class

Other values are used, but do not appear to be of relevance[1].

5.1.1.2 Code segments

Code segments contain free-form SCI bytecode. Pointers into this code are held by objects, classes, and export entries; these entries are, in turn, referenced in the export segment.

5.1.1.3 Synonym word list segments

Inside these, synonyms for certain words may be found. A synonym is a tuple (a, b), where both a and b are word groups, and b is the replacement for a if this synonym is in use. They are stored as 16 bit LE values in sequence (first a, then b). Synonyms must be set explicitly by the kernel function SetSynonyms() (as described Section 5.5.2.39). It is not possible to select synonyms selectively.

5.1.1.4 Said spec segments

This section contains said specs (explained in Section 6.2.4), tightly grouped.

5.1.1.5 String segments

This segment contains a sequence of asciiz strings describing class and object names, debug information, and (occasionally) game text.

5.1.1.6 Class segments

Classes look similar to objects:

[1] See SQ3's inventory objects for an example

`[00][01]`: Magic number 0x1234
`[02][03]`: Local variable offset (filled in at run-time)
`[04][05]`: Offset of the function selector list, relative to its own position
`[06][07]`: Number of variable selectors (= #vs)
`[08][09]`: The 'species' selector
`[0a][0b]`: The 'superClass' selector
`[0c][0d]`: The '–info–' selector
`[0e][0f]`: The 'name' selector (object/class name)
`[10]...`: (#vs-4) more variable selectors
`[08+ #vs*2][09+ #vs*2]`: Selector ID of the first varselector (0)
`[0a+ #vs*2]...`: Selector ID of the second etc. varselectors
`[08+ #vs*4][09+ #vs*4]`: Number of function selectors (#fs)
`[0a+ #vs*4]...`: Function selector code pointers
`[08+ #vs*4 + #fs*2][09+ #vs*4 + #fs*2]`: 0
`[0a+ #vs*4 + #fs*2]...`: Selector ID of the first etc. funcselectors

Simply put, they look like objects with each selector section followed by a list of selector IDs.

5.1.1.7 Export segments

External symbols are contained herein, the number of which is described by the first (16 bit LE) value in the segment. All the values that follow point to addresses that the program counter will jump to when a `calle` operation is invoked. An exception is script 0, entry 0, which points to the first object whose 'play' method should be invoked during startup (a magical entry point like C's 'main())' function).

5.1.1.8 Relocation tables

This section contains script-relative pointers pointing to pointers inside the script. These refer to script-relative addresses and need to be relocated when the script is loaded to the heap; this is done by adding the offset of the first byte of the script on the heap to each of the values referenced in this section [2].

The section itself starts with a 16 bit LE value containing the number of pointers that follow, with each of the script-relative 16 bit pointers beyond having semantics as described above

5.1.1.9 The Preload Text flag

This is an actual script section, although it is always of size 4 (i.e. only consists of the script header). It is only checked for presence; if `script.x` is loaded and contains this section, the `text.x` resource is also loaded implicitly along with it [3]

5.1.1.10 Local variable segments

This section contains the script's local variable segment, which consists of a sequence of 16 bit little-endian values.

5.1.2 Selectors

Selectors are very important in SCI. They can be either methods or object/class-relative variables, and this makes the interpretation of SCI operations like `send` a bit tricky.

Each class comes with two two-dimensional tables. The first table contains selector values and selector indices[4] for each variable selector. The second table contains selector indices and script-relative method offsets. Objects look nearly identical, but they do not contain the list of selector indices for variable selectors, since those can be looked up at the class they were instantiated from (their "species", which happens to be one of the variable selectors).

Now, whenever a selector is sent for, the engine has to determine the right action to take. FreeSCI first determines whether the selector is a variable selector, by looking for it in the list of variable selector indices of the species class of the object that the "`send`" was sent to (classes use their own class number

[2] Thanks to Francois Boyer for this information
[3] This is ignored by FreeSCI ATM, since all resources are present in memory all the time.
[4] Those can be used as an index into vocab.997, where the selector names are stored as strings.

as their species class) [5]. If it is, the selector value is either read (if no parameter was provided to the `send` call) or set (if one parameter was provided). If the selector was not part of the variable selectors of the specified object, the object's methods are checked for this selector index. If they don't contain the selector index, either, then FreeSCI recurses into checking the method selectors of the object's superclasses. If it finds the selector value there, it calls the heap address corresponding to the selector index.

5.1.3 Function invocation

SCI provides three distinct ways for invocating a function[6]:

Calling exported functions (`calle`, `callb`)
Calling selector methods (`send`, `self`, `super`)
Calling PC-relative addresses (`call`)

Exported functions are called by providing a script number and an exported function number (which is then looked up in the script's Type 7 block). They use the object they were called from to look up local variables and selectors for `self` and `super`.

Selector methods are called by providing an object and a selector index. The selector index gets looked up in the object's selector tables, and, if it is used for a method, this method gets invocated. The provided object is used for local references.

PC-relative calls only make sense inside scripts, since they jump to a position relative to the `call` opcode. The calling object is used for local references.

5.1.4 Variable types

SCI bytecode can address four types of variables (not counting the variable selectors). Those variable types are:

Local variables These are the variables stored in Type 10 script blocks. They are shared between the objects and classes of each script.

Global variables These variables are the local variables of script 0.

Temporary variables Those variables are stored on the stack. They are relative to the stack frame of the current method, so space for them must be allocated before they can be used. This is commonly done by using the `link` operation.

Parameters Parameters are stored on the stack below the current stack frame, as they technically belong to the calling function. They can be modified, if neccessary. [7]

5.2 Interpreter initialization and the main execution loop

By Lars Skovlund Version 1.0, 7. July 1999

When the interpreter initializes, it sets up a timer for 60 hertz (one that "ticks" 60 times per second). This timer does two things: it lets the so-called servers execute (most notably, the sound player and input manager) and it "feeds" the internal game clock. This 60 hz. "systick" is used all over the place. For example, it is accessible using the KGetTime kernel function. Some graphic effects depend on it, for example the "shake screen" effect. In SCI1, it is also used for timing in the palette fades. And naturally, it is used in the KWait kernel call.

Basically, the initialization proceeds as follows:

1. Initialize the heap and hunk

[5] In practice, send looks up the heap position of the requested class in a global class table.

[6] Of course, "manual" invocation (using push and jump operations) could also be used, but there are no special provisions for it, and it does not appear to be used in the existing SCI bytecode.

[7] Obviously, SCI uses a call-by-value model for primitives and call-by-reference for objects

2. Parse the config file and the command line
3. Load the drivers specified in the config file
4. Initialize the graphics subsystem.
5. Initialize the event manager
6. Initialize the window manager
7. Initialize the text parser (i.e. load the vocabulary files)
8. Initialize the music player
9. Save the machine state for restarting the game later on [8]
10. Allocate the PMachine stack on the heap.
11. Get a pointer to the game object
12. And run, by executing the play or replay method.

The right game object is found by looking in the "dispatch table" of script 0. The dispatch table has block type 7, and is an array of words. The first entry is a pointer (script relative) to the game object, for instance SQ3. If the game was restarted, the interpreter executes the replay method, play otherwise.

After looking up the address of the method in the object block, execution is started. It can be viewed as a huge switch statement, which executes con- tinuously. When the last ret statement (in the play or replay method) is met, the interpreter terminates.

The ExecuteCode function, which contains the mentioned switch statement, is called recursively. It lets other subroutines handle the object complexity, all the ExecuteCode function has is a pointer to the next in- struction. Thus, it is easy to terminate the interpreter; just return from all running instances of ExecuteCode.

So, how does an SCI program execute? Well, the play method is defined in the Game class, and it is never overridden. It consists of a huge loop which calls Game::doit continuously, followed by a pause according to the selected animation speed. That is, the script, not the interpreter, handles animation speed. Notice how the debugger very often shows the statement sag $12 upon entering the debugger? This instruction resides in Game::play, and the break occurs here because of a KWait kernel call which is executed right before that instruction. This wait takes the most execution time, so therefore the debug break is most likely to be A game programmer would then override Game::doit and place the game specific main loop here (still, Game::doit is almost identical from game to game). Execution of the Game::play main loop stops when an event causes global variable 4 to be non-zero. The last ret instruction is met, and the interpreter terminates.

5.3 The SCI Heap

SCI0 (and probably SCI1 as well) uses a heap consisting of 0xffff bytes of memory; this size corresponds to the size of one i386 real-mode memory segment minus one. [9]

5.3.1 Heap structure

The original heap starts with 200 separate entries with a size of four bytes. Each of those entries appears to be a pointer to "hunk" memory, which is separate from the heap and not covered here. The actual heap base pointer points to the first byte that is not part of these pointers.

[8] This is quite interesting, the KRestartGame kernel call is implemented using a simple setjmp/longjmp pair.
[9] This appears to be the maximum size; the games generally require less heap space.

5.3.2 Memory handles

A memory handle consists of two consecutive unsigned 16 bit integers:

- The memory block size
- The heap address of the next memory handle

in this sequence.

Memory handles are stored inside of the heap; they delimit the holes in the heap by indexing each other, with the exception of the last handle, which points to zero.

5.3.3 Initialization

The list is initialized to 0. Memory handle #0 is set to contain 0xffff minus the size required by the memory handles (800 bytes) to a total of 0xfcdf, the pointer to the next free index is set to 0x0.

5.3.4 Memory allocation

The memory allocation function takes one parameter; this is the requested allocation block size. If it is 0, the function aborts. Otherwise, the size is increased by 2 (and then again by 1, if it is odd, for alignment purposes).

After the memory allocation algorithm finds a sufficiently large memory hole, it allocates its memory by splitting the memory hole and allocating the lower part (or by swallowing the upper part if its size would be less than four). It adjusts the previous memory handle (which used to point to the start of the now allocated part of the heap) to point to the next hole, and then goes on to write its size to the first two bytes of its newly allocated home.

If no sufficiently large memory hole can be found, the function returns 0; otherwise, it returns a heap pointer to the start of the allocated block (i.e. to the two bytes that carry the block's size).

Memory deallocation does this process in reverse; it also merges adjacent memory holes to prevent memory fragmentation.

5.4 The Sierra PMachine

Lars Skovlund, Dark Minister and Christoph Reichenbach

Version 1.0, 6. July 1999

This document describes thee design of the Sierra PMachine (the virtual CPU used for executing SCI programs). It is a special CPU, in the sense that it is designed for object oriented programs.

There are three kinds of memory in SCI: Variables, objects, and stack space. The stack space is used in a Last-In-First-Out manner, and is primarily used for temporary space in a routine, as well as passing data from one routine to another. Note that the stack space is used bottom-up by the original in- terpreter, instead of the more usual top-down. I don't know if this has any significance for us.

Scripts are loaded into the PMachine by creating a memory imagee of it on the heap. For this reason, the script file format may seem a bit obscure at times. It is optimized for in-memory performance, not readability. It should be mentioned here that a lot of fixup stuff is done by the interpreter. In the script files, all addresses are specified as script-relative. These are converted to absolute offsets. The species and superClass fields of all objects are converted into pointers to the actual class etc.

There are four types of variables. These are called global, local, temporary, and parameter. All four types are simple arrays of 16-bit words. A pointer is kept for each type, pointing to the list that is currently active. In fact, only the global variable list is constant in memory. The other pointers are changed frequently, as scripts are loaded/unloaded, routines called, etc. The variables are always referenced as an index into the variable list. I'll explain the four types below - the names in parantheses will be used occasionally in the rest of the text:

5.4.1 Local variables (LocalVar)

This variable type is called "local" because it belongs to a specific script. Each script may have its own set of local variables, defined by script block type 10. As long as the code from a specific script is running, the local variables for that script are "active" (pointed to by the mentioned pointer).

5.4.2 Global variables

These, like the local variables, reside in script space (in fact, they are the local variables of script 0!). But the pointer to them remains constant for the whole duration of the program.

5.4.3 Temporary variables

These are allocated by specific subroutines in a script. They reside on the PMachine stack and are allocated by the link opcode. The temp variables are automatically discarded when the subroutine returns.

5.4.4 Parameter variables

These variables also reside on the stack. They contain information passed from one routine to another. Any routine in SCI is capable of taking a variable number of parameters, if need be. This is possible because a list size is pushed as the first thing before calling a routine. In addition to this, a frame size is passed to the call* functions.

5.4.5 Objects

While two adjacent variables may be entirely unrelated, the contents of an object is always related to one task. The object, like the variable tables, provides storage space. This storage space is called properties. Depending on the instructions used, a property can be referred to by index into the object structure, or by property IDs (PIDs). For instance, the name property has the PID 17h, but the offset 6. The property IDs are assigned by the SCI compiler, and it is the "compatible" way of accessing object data. Whereas the offset method is used only internally by an object to access its own data, the PID method is used externally by objects to read/write the data fields of other objects. The PID method is also used to call methods in an object, either by the object itself, by another object, or by the SCI inter- preter. Yes, this really happens sometimes.

5.4.6 The PMachine "registers"

The PMachine can be said to have a number of registers, although none of them can be accessed explicitly by script code. They are used/changed implicitly by the script opcodes:

Acc	the accumulator. Used for result storage and input for a number of opcodes.
IP	the instruction pointer.[10] Points to the currently executing instruction
Vars	an array of 4 values, pointing to the current variables of each mentioned type
Object	points to the currently executing object.
SP	the current stack pointer. Note that the stack in the original SCI interpreter is used bottom-up instead of the more usual top-down.

The PMachine, apart from the actual instruction pointer, keeps a record of which object is currently executing.

5.4.7 The instruction set

The PMachine CPU potentially has 128 instructions (however, a couple of these are invalid and generate an error). Some of these instructions have a flag which specify whether the opcode has byte- or word-sized operands (I will refer to this as variably-sized parameters, as opposed to constant parameters). Other instructions have only one calling form. These instructions simply disregard the operand size flag. Ideally, however, all script instructions should be prepared to take variably-sized operands. Yet another group of instructions take both a constant parameter and a variably-sized parameter. The format of an opcode byte is as follows:

bit 7-1	opcode number
bit 0	operand size flag

5.4.7.1 Relative addresses

Certain instructions (in particular, branching ones) take relative addresses as a parameter. The actual address is calculated based on the instruction after the branching instruction itself. In this example, the bnt instruction, if the branch is made, jumps over the ldi instruction.

```
eq?
bnt +2
ldi byte 2
push
```

Relative addresses are signed values.

5.4.7.2 Dispatch addresses

The callb and calle instructions take a so-called dispatch index as a parameter. This index is used to look up an actual script address, using the so-called dispatch table. The dispatch table is located in script block type 7 in the script file. It is a series of words - the first one, as in so many other places in the script file, is the number of entries.

5.4.7.3 Frame sizes

In every call instruction, a value is included which determines the size of the parameter list, as an offset into the stack. This value discounts the list size pushed by the SCI code. For instance, consider this example from real SCI code:

```
pushi 3 ; three parameters passed
pushi 4 ; the screen flag
pTos x ; push the x property
pTos y ; push the y property
callk OnControl, 6
```

Notice that, although the callk line specifies 6 bytes of parameters, the kernel routine has access to the list size (which is at offset 8)!

5.4.7.4 PErrors

These are internal errors in the interpreter. They are usually caused by buggy script code. The PErrors end up displaying an "Oops!" box in the original interpreter (it is interesting to see how Sierra likes to believe that PErrors are caused by the user - judging by the message "You did something we weren't expecting"!). In the original interpreter, specifying -d on the command line causes it to give more detailed information about PErrors, as well as activating the internal debugger if one occurs.

5.4.7.5 Class numbers and adresses

The key to finding a specific class lies in the class table. This class table resides in VOCAB.996, and contains the numbers of scripts that carry classes. If a script has more than one class defintion, the script number is repeated as necessary. Notice how each script number is followed by a zero word? When the interpreter loads a script, it checks to see if the script has classes. If it does, a pointer to the object structure is put in this empty space.

5.4.7.6 The instructions

The instructions are described below. I have used Dark Minister's text on the subject as a starting point, but many things have changed; stuff explained more thoroughly, errors corrected, etc. The first 23 instructions (up to, but not including, bt) take no parameters.

These functions are used in the pseudocode explanations:

pop(): sp -= 2; return *sp;
push(x): *sp = x; sp += 2; return x;
The following rules apply to opcodes:

1. Parameters are signed, unless stated otherwise. Sign extension is performed.
2. Jumps are relative to the posisition of the next operation.
3. *TOS refers to the TOS (Top Of Stack) element.
4. "tmp" refers to a temporary register that is used for explanation purposes only.

- op 0x00: bnot (1 byte)
  op 0x01: bnot (1 byte)

  Binary not:

  ```
  acc ^= 0xffff;
  ```

- op 0x02: add (1 byte)
  op 0x03: add (1 byte)

  Addition:

  ```
  acc += pop();
  ```

- op 0x04: sub (1 byte)
  op 0x05: sub (1 byte)

  Subtraction:

  ```
  acc = pop() - acc;
  ```

- op 0x06: mul (1 byte)
  op 0x07: mul (1 byte)

  Multiplication:

  ```
  acc *= pop();
  ```

- op 0x08: div (1 byte)
  op 0x09: div (1 byte)

  Division:

  ```
  acc = pop() / acc;
  ```

  Division by zero is caught => acc = 0.

- op 0x0a: mod (1 byte)
  op 0x0b: mod (1 byte)

  Modulo:

  ```
  acc = pop() % acc;
  ```

  Modulo by zero is caught => acc = 0.

- op 0x0c: shr (1 byte)
  op 0x0d: shr (1 byte)

  Shift Right logical:

  ```
  acc = pop() >> acc;
  ```

- op 0x0e: shl (1 byte)
  op 0x0f: shl (1 byte)

  Shift Left logical:

  ```
  acc = pop() << acc;
  ```

- op 0x10: xor (1 byte)
  op 0x11: xor (1 byte)

  Exclusive or:

  ```
  acc ^= pop();
  ```

- op 0x12: and (1 byte)
  op 0x13: and (1 byte)

  Logical and:

  ```
  acc &= pop();
  ```

- op 0x14: or (1 byte)
  op 0x15: or (1 byte)

  Logical or:

  ```
  acc |= pop();
  ```

- op 0x16: neg (1 byte)
  op 0x17: neg (1 byte)

  Sign negation:

  ```
  acc = -acc;
  ```

- op 0x18: not (1 byte)
  op 0x19: not (1 byte)

  Boolean not:

  ```
  acc = !acc;
  ```

- op 0x1a: eq? (1 byte)
  op 0x1b: eq? (1 byte)

  Equals?:

  ```
  prev = acc;
  acc = (acc == pop());
  ```

- op 0x1c: ne? (1 byte)
  op 0x1d: ne? (1 byte)

  Is not equal to?

  ```
  prev = acc;
  acc = !(acc == pop());
  ```

- op 0x1e: gt? (1 byte)
  op 0x1f: gt? (1 byte)

  Greater than?

  ```
  prev = acc;
  acc = (pop() > acc);
  ```

- op 0x20: ge? (1 byte)
  op 0x21: ge? (1 byte)

  Greater than or equal to?

  ```
  prev = acc;
  acc = (pop() >= acc);
  ```

- op 0x22: lt? (1 byte)
  op 0x23: lt? (1 byte)

  Less than?

  ```
  prev = acc;
  acc = (pop() < acc);
  ```

- op 0x24: le? (1 byte)
  op 0x25: le? (1 byte)

  Less than or equal to?

  ```
  prev = acc;
  acc = (pop() <= acc);
  ```

- op 0x26: ugt? (1 byte)
  op 0x27: ugt? (1 byte)

  Unsigned: Greater than?

  ```
  acc = (pop() > acc);
  ```

- op 0x28: uge? (1 byte)
  op 0x29: uge? (1 byte)

  Unsigned: Greather than or equal to?

  ```
  acc = (pop() >= acc);
  ```

- op 0x2a: ult? (1 byte)
  op 0x2b: ult? (1 byte)

  Unsigned: Less than?

  ```
  acc = (pop() < acc);
  ```

- op 0x2c: ule? (1 byte)
  op 0x2d: ule? (1 byte)

  Unsigned: Less than or equal to?

  ```
  acc = (pop() >= acc);
  ```

- op 0x2e: bt W relpos (3 bytes)
  op 0x2f: bt B relpos (2 bytes)

  Branch relative if true

  ```
  if (acc) pc += relpos;
  ```

- op 0x30: bnt W relpos (3 bytes)
  op 0x31: bnt B relpos (2 bytes)

  Branch relative if not true

  ```
  if (!acc) pc += relpos;
  ```

- op 0x32: jmp W relpos (3 bytes)
  op 0x33: jmp B relpos (2 bytes)

  Jump

  ```
  pc += relpos;
  ```

- op 0x34: ldi W data (3 bytes)
  op 0x35: ldi B data (2 bytes)

  Load data immediate

  ```
  acc = data;
  ```

  Sign extension is done for 0x35 if required.

- op 0x36: push (1 byte)
  op 0x37: push (1 byte)

  Push to stack

  ```
  push(acc)
  ```

- op 0x38: pushi W data (3 bytes)
  op 0x39: pushi B data (2 bytes)

  Push immediate

  ```
  push(data)
  ```

  Sign extension for 0x39 is performed where required.

- op 0x3a: toss (1 byte)
  op 0x3b: toss (1 byte)

  TOS subtract

  ```
  pop();
  ```

  For confirmation: Yes, this simply tosses the TOS value away.

- op 0x3c: dup (1 byte)
  op 0x3d: dup (1 byte)

  Duplicate TOS element

  ```
  push(*TOS);
  ```

- op 0x3e: link W size (3 bytes)
  op 0x3f: link B size (2 bytes)

  ```
  sp += (size * 2);
  ```

- op 0x40: call W relpos, B framesize (4 bytes)
  op 0x41: call B relpos, B framesize (3 bytes)

  Call inside script.

  ```
  (See description below)
  sp -= (framesize + 2 + &rest_modifier);
  &rest_modifier = 0;
  ```

  This calls a script subroutine at the relative position *relpos*, setting up the ParmVar pointer first. ParmVar points to sp-*framesize* (but see also the &rest operation). The number of parameters is stored at word offset -1 relative to ParmVar.

- op 0x42: callk W kfunct, B kparams (4 bytes)
  op 0x43: callk B kfunct, B kparams (3 bytes)

  Call kernel function (see Section 5.5)

  ```
  sp -= (kparams + 2 + &rest_modifier);
  &rest_modifier = 0;
  (call kernel function kfunct)
  ```

- op 0x44: callb W dispindex, B framesize (4 bytes)
  op 0x45: callb B dispindex, B framesize (3 bytes)

  Call base script

  ```
  (See description below)
  sp -= (framesize + 2 + &rest_modifier);
  &rest_modifier = 0;
  ```

  This operation starts a new execution loop at the beginning of script 0, public method *dispindex* (Each script comes with a dispatcher list (type 7) that identifies public methods). Parameters are handled as in the call operation.

- op 0x46: calle W script, W dispindex, B framesize (5 bytes)
  op 0x47: calle B script, B dispindex, B framesize (4 bytes)

  Call external script

```
(See description below)
sp -= (framesize + 2 + &rest_modifier);
&rest_modifier = 0;
```

This operation performs a function call (implicitly placing the current program counter on the execution stack) to an "external" procedure of a script. More precisely, exported procedure `dispindex` of script `script` is invoked, where `dispindex` is an offset into the script's Exports list (i.e., `dispindex` $= n * 2$ references the nth exported procedure).

The "Exports list" is defined in the script's type 7 object (cf. section 5.1.1). It is an error to invoke a script which does not exist or which does not provide an Exports list, or to use a dispatch index which does not point into an even address within the Exports list.

- op 0x48: ret (1 byte)
  op 0x49: ret (1 byte)

Return: returns from an execution loop started by call, calle, callb, send, self or super.

- op 0x4a: send B framesize (2 bytes)
  op 0x4b: send B framesize (2 bytes)

Send for one or more selectors. This is the most complex SCI operation (together with self and class).

Send looks up the supplied selector(s) in the object pointed to by the accumulator. If the selector is a variable selector, it is read (to the accumulator) if it was sent for with zero parameters. If a parameter was supplied, this selector is set to that parameter. Method selectors are called with the specified parameters.

The selector(s) and parameters are retreived from the stack frame. Send first looks up the selector ID at the bottom of the frame, then retreives the number of parameters, and, eventually, the parameters themselves. This algorithm is iterated until all of the stack frame has been "used up". Example:

```
; This is an example for usage of the SCI send operation
   pushi x      ; push the selector ID of x
   push1        ; 1 parameter: x is supposed to be set
   pushi 42     ; That's the value x will get set to
   pushi moveTo ; In this example, moveTo is a method selector.
   push2        ; It will get called with two parameters-
   push         ; The accumulator...
   lofss 17     ; ...and PC-relative address 17.
   pushi foo    ; Let's assume that foo is another variable selector.
   push0        ; This will read foo and return the value in acc.
   send 12      ; This operation does three quite different things.
```

- op 0x4c
  op 0x4d
  op 0x4e
  op 0x4f

These opcodes don't exist in SCI.

- op 0x50: class W function (3 bytes)
  op 0x51: class B function (2 bytes)

Get class address. Sets the accumulator to the memory address of the specified *`function`* of the current object.

- op 0x52
  op 0x53

These opcodes don't exist in SCI.

- op 0x54: self B stackframe (2 bytes)
  op 0x55: self B stackframe (2 bytes)

Send to self. This operation is the same as the send operation, except that it sends to the current object instead of the object pointed to by the accumulator.

- op 0x56: super W class, B stackframe (4 bytes)
  op 0x57: super B class, B stackframe (3 bytes)

Send to any class. This operation is the same as the send operation, except that it sends to an arbitrary `class`.

- op 0x58: &rest W paramindex (3 bytes)
  op 0x59: &rest B paramindex (2 bytes)

Pushes all or part of the ParmVar list on the stack. The number specifies the first parameter variable to be pushed. I'll give a small example. Suppose we have two functions:

function a(y,z) and function b(x,y,z)

function b wants to call function a with its own y and z parameters. Easy job, using the the normal lsp instruction. Now suppose that both function a and b are designed to take a variable number of parameters:

function a(y,z,...) and function b(x,y,z,...)

Since lsp does not support register indirection, we can't just push the variables in a loop (as we would in C). Instead this function is used. In this case, the instruction would be &rest 2, since we want the copying to start from y (inclusive), the second parameter.

Note that the values are copied to the stack *immediately*. The `&rest_modifier` is set to the number of variables pushed afterwards.

- op 0x5a: lea W type, W index (bytes)
  op 0x5b: lea B type, B index (bytes)

Load Effective Address

The variable type is a bit-field used as follows:

bit 0 unused

bit 1-2 the number of the variable list to use
0 - globalVar
2 - localVar
4 - tempVar
6 - parmVar

bit 3 unused

bit 4 set if the accumulator is to be used as additional index

Because it is so hard to explain, I have made a transcription of it here:

```
short *vars[4];

int acc;

int lea(int vt, int vi)
{
  return &((vars[(vt >> 1) &amp; 3])[vt &amp; 0x10 ? vi+acc : vi]);
}
```

- op 0x5c: selfID (1 bytes)
  op 0x5d: selfID (1 bytes)

Get 'self' identity: SCI uses heap pointers to identify objects, so this operation sets the accumulator to the address of the current object.

```
acc = object
```

- op 0x5e
  op 0x5f

  These opcodes don't exist in SCI.

- op 0x60: pprev (1 bytes)
  op 0x61: pprev (1 bytes)

  Push prev: Pushes the value of the prev register, set by the last comparison bytecode (eq?, lt?, etc.), on the stack.

```
push(prev)
```

- op 0x62: pToa W offset (3 bytes)
  op 0x63: pToa B offset (2 bytes)

  Property To Accumulator: Copies the value of the specified property (in the current object) to the accumulator. The property is specified as an offset into the object structure.

- op 0x64: aTop W offset (3 bytes)
  op 0x65: aTop B offset (2 bytes)

  Accumulator To Property: Copies the value of the accumulator into the specified property (in the current object). The property number is specified as an offset into the object structure.

- op 0x66: pTos W offset (3 bytes)
  op 0x67: pTos B offset (2 bytes)

  Property To Stack: Same as pToa, but pushes the property value on the stack instead.

- op 0x68: sTop W offset (3 bytes)
  op 0x69: sTop B offset (2 bytes)

  Stack To Property: Same as aTop, but gets the new property value from the stack instead.

- op 0x6a: ipToa W offset (3 bytes)
  op 0x6b: ipToa B offset (2 bytes)

  Incement Property and copy To Accumulator: Increments the value of the specified property of the current object and copies it into the accumulator. The property number is specified as an offset into the object structure.

- op 0x6c: dpToa W offset (3 bytes)
  op 0x6d: dpToa B offset (2 bytes)

  Decrepent Property and copy to Accumulator: Decrements the value of the specified property of the current object and copies it into the accumulator. The property number is specified as an offset into the object structure.

- op 0x6e: ipTos W offset (3 bytes)
  op 0x6f: ipTos B offset (2 bytes)

  Increment Property and push to Stack Same as ipToa, but pushes the result on the stack instead.

- op 0x70: dpTos W offset (3 bytes)
  op 0x71: dpTos B offset (2 bytes)

  Decrement Property and push to stack: Same as dpToa, but pushes the result on the stack instead.

- op 0x72: lofsa W offset (3 bytes)
  op 0x73: lofsa B offset (2 bytes)

  Load Offset to Accumulator:

```
acc = pc + offset
```

  Adds a value to the post-operation pc and stores the result in the accumulator.

- op 0x74: lofss W offset (3 bytes)
  op 0x75: lofss B offset (2 bytes)

  Load Offset to Stack:

  ```
  push(pc + offset)
  ```

  Adds a value to the post-operation pc and pushes the result on the stack.

- op 0x76: push0 (1 bytes)
  op 0x77: push0 (1 bytes)

  Push 0:

  ```
  push(0)
  ```

- op 0x78: push1 (1 bytes)
  op 0x79: push1 (1 bytes)

  Push 1:

  ```
  push(1)
  ```

- op 0x7a: push2 (1 bytes)
  op 0x7b: push2 (1 bytes)

  Push 2:

  ```
  push(2)
  ```

- op 0x7c: pushSelf (1 bytes)
  op 0x7d: pushSelf (1 bytes)

  Push self:

  ```
  push(object)
  ```

- op 0x7e
  op 0x7f

  These operations don't exist in SCI.

- op 0x80 - 0xfe: [ls+-][as][gltp]i? W index (3 bytes)
  op 0x81 - 0xff: [ls+-][as][gltp]i? B index (2 bytes)

  The remaining SCI operations work on one of the four variable types. The variable index is retreived by taking the heap pointer for the specified variable type, adding the *index* and possibly the accumulator, and executing the operation according to the following table:

  Bit 0 Used as with all other opcodes with variably-sized parameters:
  0: 16 bit parameter
  1: 8 bit parameter

  Bits 1,2 The type of variable to operate on:
  0: Global
  1: Local
  2: Temporary
  3: Parameter

  Bit 3 Whether to use the accumulator or the stack for operations:
  0: Accumulator
  1: Stack

  Bit 4 Whether to use the accumulator as a modifier to the supplied index:
  0: Don't use accumulator as an additional index
  1: Use the accumulator as an additional index

Bits 5,6 The type of execution to perform:
0: Load the variable to the accumulator or stack
1: Store the accumulator or stack in the variable
2: Increment the variable, then load it into acc or on the stack
3: Decrement the variable, then load it into acc or on the stack

Bit 7 Always 1 (identifier for these opcodes)

Example: "sagi 2" would Store the Accumulator in the Global variable indexed with 2 plus the current accumulator value (this rarely makes sense, obviously). "+sp 6" would increment the parameter at offset 6 (the third parameter, not counting the argument counter), and push it on the stack.

5.5 Kernel functions

(Acknowledgements for this section go to Lars Skovlund, Francois Boyer and Jeremy Tartaglia for providing additional information).

In SCI0, calls to the SCI kernel are initiated by using the `callk` opcode. `callk` has the opcode `0x42` or `0x43`; `0x42` takes one 16 bit little endian and one 8 bit paramter, `0x43` takes two 8 bit parameters. The first parameter is the number of the kernel function to be called, the second number undetermined (as of yet).

Opcode summary:
op 0x42: callk W kfunct, B kparams (4 bytes)
op 0x43: callk B kfunct, B kparams (3 bytes)

The number of parameters passed to the kernel function are determined by kparam. A total number of (kparams+2) bytes are removed from the local stack and passed on to the kernel function. The first two of those bytes are apparently always created by pushing the number of following bytes. For example, if Load(view, 10) is called, then we've got two word parameters, "view" (0x0080) and "10" (0x000a). So the callk function would have kparams set to 4; this value would be pushed to the stack first, followed by the two parameters. So the stack would look like this (left means lower address, byte ordering little endian):

```
02 00 80 00 0a 00
```

before calling Load().

Return values are returned into the accumulator, unless stated otherwise. If return type is stated as (void), then the accumulator is not modified.

5.5.1 Parameter types

SCI0 uses only little endian 16 bit integer values for parameters. However, this document distinguishes between different uses of those integers by defining the following variable types:

(word)	16 bit signed little endian integer
(HeapPtr)	As (word); interpreted as a pointer to a heap address
(DblList)	As (HeapPtr); interpreted as offset of a doubly linked list
(Node)	As (HeapPtr); interpreted as offset of a list node
(&FarPtr)	As (HeapPtr); interpreted as the 32 bit pointer stored at the referenced heap address
(Point)	A sequence of two (word)s to describe a point on the screen, with the y coordinate being the first in the sequence.
(Rect)	A sequence of four (word)s describing a rectangle. If you read "(Rect) foo", think "(word) foo_ymin, (word) foo_xmin, (word) foo_ymax, (word) foo_xmax" instead.
(String)	If greater than or equal to 1000, this is the heap address of a text string. If less than 1000, it is the number of a text resource, and immediately followed by another word that contains the number of the string inside the text resource.

Parameters in brackets (like "[foo]") are optional.

Most functions exit gracefully if either a NULL HeapPtr or DblList is provided.

5.5.2 SCI0 Kernel functions

5.5.2.1 Kernel function 0x00: Load(word, word)

kfunct 0x00: Load();
word ResType, word ResNr;
(word) *ResType*: The resource type number | 0x80 (as in the patch files)
(word) *ResNr*: The resource number
Returns: (&FarPtr): A HeapPtr pointing to an actual pointer on the heap.
Loads a resource. The returned HeapPtr points to a special point on the heap where a pointer (32 bits) to the memory location of the specified resource is located. If the resource type equals sci_memory, the resource number is interpreted as a memory size instead; the specified number of bytes is allocated dynamically, and a handle returned.

5.5.2.2 Kernel function 0x01: UnLoad(word, word)

kfunct 0x01: UnLoad();
word ResType, word ResNr;
(word) *ResType*: The resource type number | 0x80
(word) *ResNr*: The resource number
Returns: (void)
This function unloads a resource identified by its ResType and ResNr, NOT by the HeapPtr it has been loaded to, except for sci_memory resources, where the parameters are the memory resource type and the handle.

5.5.2.3 Kernel function 0x02:ScriptID(word, word)

kfunct 0x02: ScriptID();
word ScriptNr, word DispatchNr;
(word) *ScriptNr*: Number of the script to reference
(word) *DispatchNr*: Number of the Dispatch entry inside the script to reference
Returns: (HeapPtr): The address pointed to by the specified element of the dispatch/exports table (script block type #7)
This function returns the address pointed to by an element of a script's dispatch table.

5.5.2.4 Kernel function 0x03: DisposeScript(word ScriptNumber)

kfunct 0x03: DisposeScript();
word ScriptNumber;
(word) *ScriptNumber*
Returns: (void)
Disposes a script. Unloads it, removes its entries from the class table, and frees the associated heap memory.

5.5.2.5 Kernel function 0x04: Clone(HeapPtr)

kfunct 0x04:Clone();
HeapPtr object;
(HeapPtr) *object*: The object to clone
Returns: (HeapPtr) The address of the clone
This function clones a Class or Object by copying it as a whole and mofifying the -info- selector so that it contains 1. Objects with -info- set to 0x8000 (Classes) are stripped of their selector name area, and both Objects and Classes are stripped of the function selector area.

5.5.2.6 Kernel function 0x05: DisposeClone(HeapPtr)

kfunct 0x05: DisposeClone();
HeapPtr clone;
(HeapPtr) *clone*: The clone to dispose
Returns: (void)

Frees all memory associated with a cloned object (as produced by Clone()).

5.5.2.7 Kernel function 0x06: IsObject(HeapPtr)

kfunct 0x06: IsObject();
HeapPtr suspected_object;

(HeapPtr) *suspected_object*: The address of something that is suspected to be an object.

Returns: (int) 1 if there is an object at the specified address, 0 if not.

This function checks whether the supplied heap pointer is valid and returns 0 if not, then proceeds to testing whether an object is at the indexed heap position. If it is, 1 is returned, 0 otherwise.

5.5.2.8 Kernel function 0x07: RespondsTo(?)

5.5.2.9 Kernel function 0x08: DrawPic(word[, word, word, word])

kfunct 0x08: DrawPic();
word PicNr[, word Animation, word Flags, word DefaultPalette];

(word) *PicNr*: The resource number of the picture to draw

(word) *Animation*: One of the following animation modes:

-1: Display instantly
0: horizontally open from center
1: vertically open from center
2: open from right
3: open from left
4: open from bottom
5: open from top
6: open from edges to center
7: open from center to edges
8: open random checkboard
9: horizontally close to center, reopen from center
10: vertically close to center, reopen from center
11: close to right, reopen from right
12: close to left, reopen from left
13: close to bottom, reopen from bottom
14: close to top, reopen from top
15: close from center to edges, reopen from edges to center
16: close from edges to center, reopen from center to edges
17: close random checkboard, reopen

The animation is executed when kAnimate() (see

(word) *Flags*:
Bit 0: Clear screen before drawing
Bit 1-f: unknown, probably unused

If not specified, it defaults to 1. Some interpreter versions

(word) *DefaultPalette*: The default palette number to use for drawing

Returns: (void)

The second parameter does not appear to affect anything. In QfG1, it appears to be set to 0x64 constantly. DefaultPalette is used to differentiate between day and night in QfG1. Palette 1 is used for ”night” pictures, Palette 0 for ”day” pictures there. The picture is drawn to the background image (which is used for restauration of everything with the exception of the mouse pointer). To bring it to the foreground, Animate() must be used.

5.5.2.10 Kernel function 0x09: Show()

kfunct 0x09: Show();
; Returns: (void)

Sets the PicNotValid flag to 2.

5.5.2.11 Kernel function 0x0a: PicNotValid([word])

kfunct 0x0a: PicNotValid();
[(word) NewPicNotValid];

[(word) *NewPicNotValid*]: The new value of the ”PicNotValid” flag.

Returns: (word): The previous value of the ”PicNotValid” flag

This sets the PicNotValid flag that determines whether or not the current background picture should be considered "valid" by the other kernel functions.

5.5.2.12 Kernel function 0x0b: Animate([DblList], [word])

kfunct 0x0b: Animate();
[DblList ViewList], [word cycle];
[(DblList) `ViewList`]: List of views that are to be drawn on top of the background picture
(word) <unknown>
Returns: (void)
This function draws a background picture plus some views to the foreground. If the background picture had not been drawn previously, it is animated with the animation style set during kDrawPic (see Section 5.5.2.9). Drawing the views is a rather complex issue. Refer to Section 6.3 for its description.

5.5.2.13 Kernel function 0x0c: SetNowSeen(DblList)

?? kfunct 0x0c: SetNowSeen();
DblList ViewList;
(DblList) `ViewList`: List of affected views
Returns: (void)

5.5.2.14 Kernel function 0x0d: NumLoops(HeapPtr)

kfunct 0x0d: NumLoops();
HeapPtr object;
(HeapPtr) `object`: The object which the view selector should be taken from
Returns: (word) The number of loops in the view
This function looks up the view selector in the specified object, loads the view resource associated with it, and checks for the number of animation loops in the view.

5.5.2.15 Kernel function 0x0e: NumCels(HeapPtr)

kfunct 0x0e: NumCels();
HeapPtr object;
HeapPtr `object`: The object which the selectors should be taken from
Returns: (word) The number of cels in the loop
This function looks up one specific loop in a specific view (both are taken from selectors with the same name from the object pointed to by the parameter) and returns the number of cels (animation frames) in it.

5.5.2.16 Kernel function 0x0f: CelWide(word view, word loop, word cel)

kfunct 0x0f: CelWide();
word view, word loop, word cel;
(HeapPtr) `view`: The view we're searching in `loop`: The loop the cel is contained in `cel`: The cel we're intereste
Returns: (word) The width of the cel identified by the tuple (view, loop, cel).

5.5.2.17 Kernel function 0x0f: CelWide(word view, word loop, word cel)

kfunct 0x10: CelHigh();
word view, word loop, word cel;
(HeapPtr) `view`: The view we're searching in `loop`: The loop the cel is contained in `cel`: The cel we're intereste
Returns: (word) The height of the cel identified by the tuple (view, loop, cel).

5.5.2.18 Kernel function 0x11: DrawCel(word, word, word, Point, word)

kfunct 0x11: DrawCel();
word view, word loop, word cel, Point pos, word priority;
(word) view: Number of the view resource to display
(word) loop: Number of the loop in the view resource to display
(word) cel: Number of the cel inside the loop to display
(Point) pos: Position the cel should be drawn to
(word) priority: Priority to draw the cel with
Returns: (void)
Explicitly draws a cel, specified by the complete tuple (view, loop, cel), to a specified position. Invalid loop/cel values are assumed to be 0.

5.5.2.19 Kernel function 0x12: AddToPic(DblList)

kfunct 0x12: AddToPic();
DblList picviews;
(DblList) *picviews*: A doubly linked list of PicViews, i.e. objects that are drawn statically onto the background
Returns: (void)
This function stores the list of PicViews for later use by the Animate() syscall. See Section 5.5.2.12 for more details.

5.5.2.20 Kernel function 0x13: NewWindow(Rect, HeapPtr, word, word, word, word)

kfunct NewWindow();
Rect Boundaries, HeapPtr Title, word Flags, word Priority, word FGColor, word BGColor;
(Rect) *Boundaries*: The bounding rectangle of the window
(HeapPtr) *Title*: A pointer to the window title
(word) *Flags*:
bit 0 - transparency
bit 1 - window does not have a frame
bit 2 - the window has a title (starting 10 pixels above the minimum y position specified as the
bit 3-6 - unused
bit 7 - don't draw anything
(word) *Priority*: The priority at which the window should be drawn, or -1 to force on-top drawing
(word) *FGColor*: The foreground color for the window
(word) *BGColor*: The background color
Returns: (HeapPtr): The position of the window structure on the heap
This function creates a window (see also Section 3.12), sets this window as the active port, draws the window (if neccessary), and returns with the window's heap address.

5.5.2.21 Kernel function 0x14: GetPort()

kfunct 0x14: GetPort();
; Returns: (HeapPtr): A pointer to a record with the internal representation of the currently active port.
Returns a heap pointer to a port structure.

5.5.2.22 Kernel function 0x15: SetPort()

kfunct 0x15: SetPort();
HeapPtr NewPort;
(HeapPtr) *NewPort*: The new port to set
Returns: (void)
This selects the new port which many kernel functions will draw to.
If 0 is passed, the window manager port is selected. The picture window is not accessible using this call. Only other kernel calls like KDrawPic may activate the picture window - and they always save the old port and restore it before they return.

5.5.2.23 Kernel function 0x16: DisposeWindow(HeapPtr Window)

kfunct 0x16: DisposeWindow();
HeapPtr Window;
(HeapPtr) *Window*: The heap address of the window to destroy
Returns: (void)
Destroys a window and frees the associated heap structure.

5.5.2.24 Kernel function 0x17: DrawControl(HeapPtr)

kfunct 0x17: DrawControl();
HeapPtr Control;
(HeapPtr) *Control*: The heap address of the Control to draw
Returns: (void)
This function draws a Control (see Section 3.12 for details). Please note that the correct port must be selected beforehand.

5.5.2.25 Kernel function 0x18: HiliteControl(HeapPtr)

kfunct 0x18: HiliteControl();
HeapPtr Control;
(HeapPtr) *Control*: The control to highlight
Returns: (void)
This function is used to highlight a control by drawing it with an inverted color scheme. It requires the correct port to be set beforehand. See Section 3.12 for details on the windowing Control system.

5.5.2.26 Kernel function 0x19: EditControl(HeapPtr)

kfunct 0x19: EditControl();
HeapPtr Control, HeapPtr Event;
(HeapPtr) *Control*: A heap pointer to the Control to edit
(HeapPtr) *Event*: The event to interpret
Returns: (void)
This function will apply the event provided to edit a type 3 (Edit window) Control (see Section 3.12 for a description of the control system). Normal keypresses are added to the area pointed to by Control::text, unless the total string length would be greater than Control::max. Cursor keys, backspace and a few other keys may be used to manipulate the control. In FreeSCI, some of the libreadline control keys can be used to edit and move the cursor as well. If it is called to edit a Control which is not of type 3, it returns without error. Please note that the correct port (usually the window which the Control was drawn in) must be selected beforehand.

5.5.2.27 Kernel function 0x1a: TextSize(HeapPtr, HeapPtr, word[, word])

kfunct 0x1a: TextSize();
HeapPtr dest, HeapPtr src, word font[, word maxwidth];
(HeapPtr) *dest*: The destination to write the rectangle to
(HeapPtr) *src*: A pointer to the string to analyze
(word) *font*: The number of the font resource to use for this check
(word) *maxwidth*: The maximum width to allow for the text (defaults to 192)
Returns: (void)
This function calculates the width and height the specified text will require to be displayed with the specified font and the specified maximum width. The result will be written to the (you guessed it) specified destination on the heap. The result is a rectangle structure: The first four bytes equal to zero, the next word is the height, and the last word is the width.

5.5.2.28 Kernel function 0x1b: Display(String, word...)

kfunct 0x1b: Display();
String text, word commands...;

(String) *text*: The text to work with
(word) *commands...*: A sequence of commands with parameters:
100: 2 params, (X,Y) coord of where to write on the port.
101: 1 param, -1, 0 or 1 (align right (-1), left (0) or center (1)
102: 1 param, set the text color.
103: 1 param, set the background color (-1 to draw text with transparent background)
104: 1 param, set the "gray text" flag (1 to draw disabled items)
105: 1 param, (resource number) set the font
106: 1 param, set the width of the text (the text wraps to fit in that width)
107: no param, set the "save under" flag, to save a copy of the pixels before writing the text (the handle to the
108: 1 param, (handle to stored pixels) restore under. With this command, the text and all other parameters are
Returns: (void) or (&FarPtr)(see above)
This function executes the specified commands, then draws the supplied text to the active port (unless command 108 was executed).

5.5.2.29 Kernel function 0x1c: GetEvent(word, HeapPtr)

kfunct 0x1c: GetEvent();
word Flags, HeapPtr Event;
(word) *Flags*: A bitfield: bit 0 - 14: Bit mask for the events to be returned.
bit 15: Disable joystick polling
(HeapPtr) *Event*: An Object on the stack which the results are written to.
Returns: (word): 0 if a null event was created, 1 otherwise.
This function fills an Event object with data from the event queue. The results are written to the "type", "message" and "modifiers" selectors. See Section 6.1 for details.

5.5.2.30 Kernel function 0x1d: GlobalToLocal(HeapPtr Event)

kfunct 0x1d: GlobalToLocal();
HeapPtr Event;
(HeapPtr) *Event*: pointer to the Event object to convert
Returns: (void)
This function converts a screen-relative event to a port-relative one, using the currently active port.

5.5.2.31 Kernel function 0x1e: LocalToGlobal(HeapPtr Event)

kfunct 0x1e: LocalToGlobal();
HeapPtr Event;
(HeapPtr) *Event*: pointer to the Event object to convert
Returns: (void)
This function converts a port-relative event to a screen-relative one, using the currently active port.

5.5.2.32 Kernel function 0x1f: MapKeyToDir(HeapPtr Event)

kfunct 0x1f: MapKeyToDir();
HeapPtr Event;
(HeapPtr) *Event*: pointer to the Event object to convert
Returns: (HeapPtr): A pointer to the converted object
This function converts a keyboard event to a movement event, if possible. Otherwise, the function returns without error. See Section 6.1 for details.

5.5.2.33 Kernel function 0x20: DrawMenuBar(word)

kfunct 0x20: DrawMenuBar();
word mode;
(word) *mode*: 1 to draw, 0 to clear
Returns: (void)
Either draws or clears (overdraws with black) the menu bar.

5.5.2.34 Kernel function 0x21: MenuSelect(HeapPtr[, word])

kfunct 0x21: MenuSelect();
HeapPtr event[, word flag];
(HeapPtr) event: The event to interpret
(word) flag: (unknown)
Returns: (word) The menu index of a selected option, -1 if no menu option was selected, or 0 if the event passed through all of the menu system's filters.
This function interprets the event passed to it by running several checks. First, it tries to determine whether the menu system was activated by pressing the ESC key or clicking on the menu bar. In this case, the interpreter takes over and waits for the player to select a menu option. It then returns the menu option selected (menu number, starting at 1, in the upper 8 bits, item number, starting at 1 as well, in the lower part) or -1 if no active menu item was selected. In any case, the event is claimed. If the menu system was not activated by the event, it checks the event against the key commands or Said Blocks associated with each menu entry. If there is a match, the menu coordinate tuple is returned and the event is claimed, otherwise, 0 is returned.

5.5.2.35 Kernel function 0x22: AddMenu(HeapPtr, HeapPtr)

kfunct 0x22: AddMenu();
HeapPtr title, HeapPtr content;
(HeapPtr) *title*: The menu title
(HeapPtr) *content*: The menu options
Returns: (void)
This function adds a menu to the menu bar. The menu title is passed in the first parameter, the second parameter contains a heap pointer to the menu entries. They are contained in one single string; the following special characters/character combinations are used:
'': Right justify the following text
':': Menu item separator
"–!": Seperation line: This menu item is just a separator
'#': Function key. This is replaced by an F for displaying
'^': Control key. This is replaced by \001 (CTRL) for displaying

5.5.2.36 Kernel function 0x23: DrawStatus(HeapPtr)

kfunct 0x23: DrawStatus();
HeapPtr text;
(HeapPtr) *text*: The text to draw
Returns: (void)
Draws the specified text to the title bar

5.5.2.37 Kernel function 0x24: Parse(HeapPtr, HeapPtr)

kfunct 0x24: Parse();
HeapPtr event, HeapPtr input;
(HeapPtr) *event*: The event to generate
(HeapPtr) *input*: The input line to parse
Returns: (word) 1 on success, 0 otherwise
This function parses the input line and generates a parse event (type 0x80). See Section 6.2 and Section 6.1 for details.

5.5.2.38 Kernel function 0x25: Said(HeapPtr)

kfunct 0x:25: Said();
HeapPtr said_block;
(HeapPtr) *said_block*: Pointer to a Said block
Returns: (word) 1 if the line last parsed meets the criteria of the supplied said_block, 0 otherwise.
This function is only invoked after Parse() was called, and works on output generated by this function. See Section 6.2 and Section 6.1 for details.

5.5.2.39 Kernel function 0x26: SetSynonyms(DblList)

kfunct 0x26: SetSynonyms();
HeapPtr list;
(DblList) list: List of script objects to examine
Returns: (void)
This function sets the synonyms used by the parser. Synonyms are used to replace specified word groups with other word groups. The list contains a collection of script objects; all synonyms defined by the corresponding script (which can be identified by evaluating the 'number' selector of the script object) are added to the list of active synonyms.

5.5.2.40 Kernel function 0x27: HaveMouse()

kfunct 0x27: HaveMouse();
;
Returns: (word) 1 if a mouse is available, 0 if not.
This function simply returns a flag containing the availability of a pointing device.

5.5.2.41 Kernel function 0x28: SetCursor(word, word[, Point])

kfunct 0x28: SetCursor();
word resource, word visible[, Point coordinates];
(word) `resource`: The cursor resource to use for drawing the mouse pointer
(word) `visible`: 1 if the mouse pointer should be visible, 0 if not
(Point) `coordinates`: The coordinates (relative to the wm-port) to move the mouse pointer to
Returns: (void)
This function can change the appearance and position of the mouse pointer. If no position is provided, the position remains unchanged.

5.5.2.42 Kernel function 0x29: FOpen(String, word)

kfunct 0x29: FOpen();
String fname, word mode;
(String) fname: The file name
(word) mode: The mode to open the file with
Returns: (word) a file handle on success, -1 on error
Tries to open or create a file in the CWD with the specified file name. The following modes are valid:
0: open or create: Try to open file, create it if it doesn't exist
1: open or fail: Try to open file, abort if not possible
2: create: Create the file, destroying any content it might have had

5.5.2.43 Kernel function 0x2a: FPuts(word, String)

kfunct 0x2a: FPuts();
word filehandle, String data;
(word) filehandle: Handle of the file to write to
(String) data: The string to write to the file
Returns: (void)
Writes a zero-terminated string to a file

5.5.2.44 Kernel function 0x2b: FGets(String, word, word)

kfunct 0x2b: FGets();
String dest, word maxsize, word handle;
(String) dest: Pointer to the destination buffer
(word) maxsize: Maximum number of bytes to read
(word) handle: Handle of the file to read from
Returns: (word) The number of bytes actually read

5.5.2.45 Kernel function 0x2c: FClose(word)

kfunct 0x2c: FClose();
word filehandle;
(word) filehandle: Handle of the file to close
Returns: (void)
Closes a previously opened file.

5.5.2.46 Kernel function 0x2d: SaveGame(String, word, String, String)

kfunct 0x2d: SaveGame();
String game_id, word save_nr, String save_description, String version;
(String) game_id: The game object's ID string (e.g. "SQ3")
(word) save_nr: "slot" the game is to be saved to
(String) save_description: String description of the game
(String) version: Stringified game version number
Returns: (word) 1 on success, 0 if an error occured while saving
This function saves the game state (heap, windows, call stack, view list, sound state etc.) to the savegame with the numeric id `save_nr` and the description `save_description`. `game_id` and `version` are stored alongside, for verification when the game state is restored.

5.5.2.47 Kernel function 0x2e: RestoreGame(String, word, String)

kfunct 0x2e: RestoreGame();
String game_id, word save_nr, String version;
(String) game_id: The game object's ID string
(word) save_nr: Number of the save game to restore
(String) version: The game object's version number
Returns: (void)
This function restores a previously saved game. It should only return if restoring failed.

5.5.2.48 Kernel function 0x2f: RestartGame()

kfunct 0x2f: RestartGame();
; Returns: never
If this function is invoked, the following things happen:
The restarting flag is set
The menu bar structure is destroyed
All sounds are stopped
All scripts are removed from the script table
The heap status is reset, but the heap is not cleared
After this is done, the engine restarts at a certain point (see Section 5.2), re-initializes the stack, and executes the replay method of the game object.

5.5.2.49 Kernel function 0x30: GameIsRestarting()

kfunct 0x30: GameIsRestarting();
;
Returns: (word) 1 if the game is restarting, 0 if not

5.5.2.50 Kernel function 0x31: DoSound(word, ...])

kfunct 0x31: DoSound();
word action, ...;
(word) action: The sound command subfunction number
Returns: (see below)
'action' may be one of the following:

0x0: INIT
0x1: PLAY
0x2: NOP
0x3: DISPOSE
0x4: SET_SOUND
0x5: STOP
0x6: SUSPEND
0x7: RESUME
0x8: VOLUME
0x9: UPDATE
0xa: FADE
0xb: CHECK_DRIVER
0xc: ALL_STOP
See individual descriptions below for more information.

5.5.2.51 Kernel function 0x31: DoSound(INIT, Object)

kfunct 0x31: DoSound();
word 0, Object sound_obj;
(word) 0: subfunction identifier
(Object) sound_obj: The sound object affected
Returns: (void)
Initializes the specified sound object. This will set the 'status' selector of the object to 1 ('initialized'), and load the sound indicated by the 'number' selector into the sound driver.

5.5.2.52 Kernel function 0x31: DoSound(PLAY, Object)

kfunct 0x31: DoSound();
word 1, Object sound_obj;
(word) 1: The subfunction identifier
(Object) sound_obj: The sound object affected
Returns: (void)
Starts to play the song represented by the specified sound object. This will also set the 'status' selector of the object to 2 ('playing').

5.5.2.53 Kernel function 0x31: DoSound(NOP)

kfunct 0x31: DoSound();
word 2;
(word) 2: The sound command subfunction number
Returns: (void)
No action appears to be associated with this subfunction call.

5.5.2.54 Kernel function 0x31: DoSound(DISPOSE, Object)

kfunct 0x31: DoSound();
word 3, Object sound_obj;
(word) 3: The sound command subfunction number
(Object) sound_obj: The sound object affected
Returns: (void)
Removes the song indexed by a sound object from the sound server song list

5.5.2.55 Kernel function 0x31: DoSound(SET_SOUND, word)

kfunct 0x31: DoSound();
word 4, word state;
(word) 4: The sound command subfunction number
(word) state: 1 if sound should be active, 0 if it should be turned off
Returns: (word) 1 if currently active, 0 if currently muted.

This function completely mutes or un-mutes the sound subsystem. If called with no parameters, it returns the current status.

5.5.2.56 Kernel function 0x31: DoSound(STOP, Object)

kfunct 0x31: DoSound();
word 5, Object sound_obj;
(word) 5: The sound command subfunction number
(Object) sound_obj: The sound object affected
Returns: (void)
Stops playing the song represented by the specified sound object. This will set the object's 'state' selector to 0 ('stopped').

5.5.2.57 Kernel function 0x31: DoSound(SUSPEND, Object)

kfunct 0x31: DoSound();
word 6, Object sound_obj;
(word) 6: The sound command subfunction number
(Object) sound_obj: The sound object affected
Returns: (void)
Suspends the song associated with the specified sound object. Its state is buffered, so that it can be resumed later on. The sound object's 'state' selector is set to 3 ('suspended').

5.5.2.58 Kernel function 0x31: DoSound(RESUME, Object)

kfunct 0x31: DoSound();
word 7, Object sound_obj;
(word) 7: The sound command subfunction number
(Object) sound_obj: The sound object affected
Returns: (void)
Resumes a previously suspended song. The 'state' selector is set to 2 ('playing').

5.5.2.59 Kernel function 0x31: DoSound(VOLUME[, word])

kfunct 0x31: DoSound();
word 8[, word volume];
(word) 8: The sound command subfunction number
(word) volume: An optional volume parameter
Returns: (word) The currently set sound volume (0 to 0xf)
This subfunction retrieves and returns the current sound volume. If a second parameter is supplied the volume will be set to the value of this parameter.

5.5.2.60 Kernel function 0x31: DoSound(UPDATE, Object])

kfunct 0x31: DoSound();
word 9, Object sound_obj;
(word) 9: The sound command subfunction number
(Object) sound_obj: The sound object affected
Returns: (void)
Notifies the sound server that a sound object was modified. The song priority and number of loops (stored in the 'priority' and 'loop' selectors, respectively) are re-evaulated by the sound system.

5.5.2.61 Kernel function 0x31: DoSound(FADE, Object])

kfunct 0x31: DoSound();
word 0xa, Object sound_obj;
(word) 0xa: The sound command subfunction number
(Object) sound_obj: The sound object affected
Returns: (void)

Fades the specified song. Fading takes approximately two seconds. The song status is set to 'stopped' (0) afterwards.

5.5.2.62 Kernel function 0x31: DoSound(CHECK_DRIVER)

kfunct 0x31: DoSound();
word 0xb;
(word) 0xb: The sound command subfunction number
Returns: (word) 1 if the sound driver was installed successfully, 0 if not

5.5.2.63 Kernel function 0x31: DoSound(ALL_STOP)

kfunct 0x31: DoSound();
word 0xc;
(word) 0xc: The sound command subfunction number
Returns: (void)
Stops all music and sound effects.

5.5.2.64 Kernel function 0x32: NewList()

kfunct 0x32: NewList();
;
Returns: (DblList) The address of a new node list on the heap
This function allocates and initializes a node list containing no elements.

5.5.2.65 Kernel function 0x33: DisposeList(DblList)

kfunct 0x33: DisposeList();
NodeList list;
(NodeList) *list*: The list to dispose
Returns: (void)
Frees all memory associated to a list

5.5.2.66 Kernel function 0x34: NewNode(word, word)

kfunct 0x34: NewNode();
word value, word key;
(word) *value*: The node value
(word) *key*: The node key (used for searching the list)
Returns: (Node) A new node
This function allocates a new node and initializes it with the key and value passed as parameters.

5.5.2.67 Kernel function 0x35: FirstNode(DblList)

kfunct 0x35: FirstNode();
DblList list;
(DblList) *list*: The list to examine
Returns: (Node) The first node of the list, or 0 if the list is empty

5.5.2.68 Kernel function 0x36: LastNode(DblList)

kfunct 0x36: LastNode();
DblList list;
(DblList) *list*: The list to examine
Returns: (Node) The last node of the list, or 0 if the list is empty

5.5.2.69 Kernel function 0x37: EmptyList(DblList)

kfunct 0x37: EmptyList();
DblList list;
(DblList) list: The list to check
Returns: (int) 1 if `list` is an empty list, 0 if it isn't.

5.5.2.70 Kernel function 0x38: NextNode(Node)

kfunct 0x38: NextNode();
Node node;
(Node) `node`: The node whose succcessor is to be found
Returns: (Node) The node following the supplied node, or 0 if none is available

5.5.2.71 Kernel function 0x39: PrevNode(Node)

kfunct 0x39: PrevNode();
Node node;
(Node) `node`: The node whose predecessor is to be determined
Returns: (Node) The supplied node's predecessor, or 0 if the node has no predecessor

5.5.2.72 Kernel function 0x3a: NodeValue(Node)

kfunct 0x3a: NodeValue();
Node node;
(Node) `node`: The node whose value is to be determined
Returns: (word) The value associated with the specified node

5.5.2.73 Kernel function 0x3b: AddAfter(DblList, Node, Node)

kfunct 0x3b: AddAfter();
DblList list, Node ref_node, Node new_node;
(DblList) list: The list to insert into
(Node) ref_node: The node in `list` to insert after
(Node) new_node: The node to insert
Returns: (void)
This function inserts `new_node` into `list` as the immediate successor of `ref_node`.

5.5.2.74 Kernel function 0x3c: AddToFront(DblList, Node)

kfunct 0x3c: AddToFront();
DblList list, Node node;
(DblList) `list`: The list the node is to be added to
(Node) `node`: The node to add
Returns: (void)
This function adds a node to the beginning of a doubly linked list.

5.5.2.75 Kernel function 0x3d: AddToEnd(DblList, Node)

kfunct 0x3d: AddToEnd();
DblList list, Node node;
(DblList) `list`: The list to add the node to
(Node) `node`: The node to add to the list
Returns: (void)
This function adds the specified node to the end of the specified list.

5.5.2.76 Kernel function 0x3e: FindKey(DblList, word)

kfunct 0x3e: FindKey();
DblList list, word key;
(DblList) `list`: The list in which the key is to be sought
(word) `key`: The key to seek
Returns: (Node) The node containing the key, or 0 if no node contains it
This function searches for a specific key in the nodes of a doubly linked list.

5.5.2.77 Kernel function 0x3f: DeleteKey(DblList, word)

kfunct 0x3f:();
DblList list, word key;
(DblList) `list`: The list to examine
(word) `key`: The key to find
Returns: (void)
This function searches in the supplied list for the specified key and removes the node containing it, if any can be found.

5.5.2.78 Kernel function 0x40: Random(word, word)

kfunct 0x40: Random();
word min, word max;
(word) `min`: The minimum result
(word) `max`: The maximum result
Returns: (word) A random number between min and max (inclusive)

5.5.2.79 Kernel function 0x41: Abs(word)

kfunct 0x41: Abs();
word value;
(word) `value`: The value to absolutize
Returns: (word) The absolute value of the specified parameter
This function interprets the supplied value as a signed value and returns its absolute value.

5.5.2.80 Kernel function 0x42: Sqrt(word)

kfunct 0x42: Sqrt();
word value;
(word) `value`: The value to draw the square root out of
Returns: (word) The square root of the supplied value

5.5.2.81 Kernel function 0x43: GetAngle(Point, Point)

kfunct 0x43: GetAngle();
Point origin, Point destination;
(Point) `origin`: The point to look from
(Point) `destination`: The point to look to
Returns: (word) A positive angle between the two points, relative to the screen coordinate axis.
This function returns approximately the following value: -(180.0/PI * atan2(destination.y - origin.y, destination.x - origin.x)) + 180; Where atan2(double, double) is the libm function.

5.5.2.82 Kernel function 0x44: GetDistance(Point, Point)

kfunct 0x44: GetDistance();
Point foo, Point bar;
(Point) `foo`: A point in two-dimensional integer space
(Point) `bar`: Another two-dimensional integer point
Returns: (int) The euklidian distance between the points foo and bar

5.5.2.83 Kernel function 0x45: Wait(word)

kfunct 0x45: Wait();
word ticks;
(word) *ticks*: The number of game ticks (60 Hz beats) to wait
Returns: (word) The time passed in between the finish of the last Wait() syscall

5.5.2.84 Kernel function 0x46: GetTime([word])

kfunct 0x46: GetTime();
word mode;
(wrod) *mode*: If this parameter is supplied, the time of day is returned.
Returns: (word) Either the time of day in seconds, or the elapsed number of ticks since the interpreter started.
This function is somewhat strange, because it determines its behaviour not by the value of a parameter passed, but by its presence instead. Please note that the time of day in this case does not distinguish between am and pm.

5.5.2.85 Kernel function 0x47: StrEnd(HeapPtr)

kfunct 0x47: StrEnd();
HeapPtr string;
(HeapPtr) *string*: The string whose terminator should be found
Returns: (HeapPtr) The address of the null terminator of the indexed string

5.5.2.86 Kernel function 0x48: StrCat(HeapPtr, HeapPtr)

kfunct 0x48: StrCat();
HeapPtr dest, HeapPtr source;
(HeapPtr) *dest*: The string whose end is appended to
(HeapPtr) *source*: The string to append
Returns: (HeapPtr) dest
This function concatenates two strings on the heap.

5.5.2.87 Kernel function 0x49: StrCmp(HeapPtr, HeapPtr[, word])

kfunct 0x49: StrCmp();
HeapPtr foo, HeapPtr bar[, word length];
(HeapPtr) *foo*: The one string to compare
(HeapPtr) *bar*: The other string to compare
(int) *width*: The maximum number of characters to compare
Returns: (word) -1 if foo is less than bar, 0 if both are equal, 1 if foo is greater than bar
This function simply encapsulates the libc `strcmp(char *, char *)` and `strncmp(char *, char *, int)` functions.

5.5.2.88 Kernel function 0x4a: StrLen(HeapPtr)

kfunct 0x4a: StrLen();
HeapPtr string;
(HeapPtr) *string*: The string whose length should be calculated
Returns: (word) The length of the specified string.

5.5.2.89 Kernel function 0x4b: StrCpy(HeapPtr, HeapPtr[, word])

kfunct 0x4b: StrCpy();
HeapPtr dest, HeapPtr src[, word length];
(HeapPtr) *dest*: The destination to copy the string to
(HeapPtr) *src*: The source from which the string is to be copied
(word) *length*: The maximum length of the string to copy
Returns: (HeapPtr) dest

Copies a string, plus the trailing \0 terminator. The length of the string may be reduced with the optional length parameter. This function simply encapsulates the libc `strcpy(char *, char *)` and `strncpy(char *, char *, int)` fucntions.

5.5.2.90 Kernel function 0x4c: Format(HeapPtr, String,...)

kfunct 0x4c: Format();
HeapPtr dest, String format, parameters...;
(HeapPtr) *dest*: The heap destination to write to
(String) *format*: The format to use
(misc) *parameters*: The values and strings to insert
Returns: (HeapPtr) dest
This syscall acts as a frontend to the libc `sprintf(char *, char *)` function.

5.5.2.91 Kernel function 0x4d: GetFarText(word, word, HeapPtr)

kfunct 0x4d: GetFarText();
word resnr, word stringnr, HeapPtr dest;
(word) *resnr*: Number of the text resource to retreive the text from
(word) *stringnr*: Number of the string inside the resource to retreive
(HeapPtr) *dest*: The destination to write the text to
Returns: (HeapPtr) dest
Retreives a string from a text resource and puts it on the heap.

5.5.2.92 Kernel function 0x4e: ReadNumber(HeapPtr)

kfunct 0x4e: ReadNumber();
HeapPtr src;
(HeapPtr) *src*: The address of the string to interpret as a number
Returns: (word) The numeric value of the supplied string
This function acts as a frontend to the libc `atoi(char *)` function, with one exception: Numbers beginning with a '$' are interpreted as hexadecimal numbers.

5.5.2.93 Kernel function 0x4f: BaseSetter(HeapPtr)

kfunct 0x4f: BaseSetter();
HeapPtr view_obj;
(HeapPtr) view_obj: The view object whose base is to be set
Returns: (void)
This method is used to set the bounding rectangle of a view. The bounding rectangle is specified by the set {brLeft, brRight, brTop, brBottom} of selectors, which indicate the window-relative boundary points of the object's bounding rectangle. The rectangle defined here is used for colission detection, among other things.

The algorithm employed by FreeSCI to determine these values appears to be either identical or very close to the original algorithm; it depends several of the object's selectors (x, y, z, ystep, view) the width and height of the view indicated by its (view, loop, cel) selectors, and that view's horizontal and vertical pixel offset modifyers (xmod, ymod). The algorithm works as follows:

```
brLeft := x - xmod - width / 2
brRight := brLeft + width
brBottom := y - z - ymod + 1
brTop := brBottom - ystep
```

5.5.2.94 Kernel function 0x50: DirLoop(HeapPtr, word)

kfunct 0x50: DirLoop();
HeapPtr object, word angle;

(HeapPtr) object: The object whose loop selector is to be set
(word) angle: The angle which is to be used as a base to choose the loop angle
Returns: (void)
This function sets the loop selector of the specified object to a value implied by the 'angle' parameter, according to the following table:

angle	loop value
angle < 45 || angle >= 314	3
angle >= 45 && angle < 135	0
angle >= 135 && angle < 225	2
angle >= 225 && angle < 314	1

5.5.2.95 Kernel function 0x51: CanBeHere(HeapPtr [, DblList])

kfunct 0x51: CanBeHere();
HeapPtr obj [, DblList clip_list];
(HeapPtr) obj: The object to test
(DblList) clip_list: An optional list of objects to test *obj* against
Returns: (int) 1 if *obj* can be where it is, 0 if not.
This function first retreives *obj*'s signal and illegalBits selectors, plus its brRect (boundary rectangle, consisting of brTop, brBottom, brLeft and brRight). If either of the DONT_RESTORE or IGNORE_ACTOR flags is set, the function returns 1, otherwise it proceeds with verifying that
illegalBits bitwise-AND the disjunction of all elements of $\{2^n | \exists$ a pixel with the color value n inside the control
$\nexists \text{pic} \in \text{clip\_list}.(\text{pic::signal}\&(\text{DONT\_RESTORE} \mid \text{IGNORE\_ACTOR})) = 0 \wedge \text{pic::brRect} \cap \text{obj::brRect} \neq \emptyset$
If both conditions are met, 1 is returned. Otherwise, 0 is returned.

5.5.2.96 Kernel function 0x52: OnControl(word, Point | Rect)

kfunct 0x52: OnControl();
word map, Point|Rect area;
(word) map: The map to check (bit 0: visual, bit 1: priority, bit 2: special)
(Point) or (Rect) Area: The point or rectangle that is to be scanned
Returns: (word) The resulting bitfield
This function scans the indicated point or area on the specified *map*, and sets the bit corresponding to each color value found correspondingly. For example, if scanning map 4 (special) would touch two areas, one with color value 1 and one with color value 10, the resulting return value would be 0x0402 (binary 0000010000000010). See also Section 3.13.

5.5.2.97 Kernel function 0x53: InitBresen(HeapPtr [, word])

kfunct 0x53: InitBresen();
HeapPtr mover, word step_factor;
(HeapPtr) mover: The mover object to initialize
(word) step_factor: A factor to multiply the step size with (defaults to 1)
Returns: (void)
Initializes a mover object for bresenham movement from the object's client's coordinates to the coordinates specified by its own pair of (x,y) selectors. To do this, it retreives the mover's client, and calculates the result values according to the algorithm for determining the initial values for iterative line drawing according to the Bresenham line algorithm:

```
client := mover::client
dx := mover::x - client::x
dy := mover::y - client::y

vxmax := client::xStep * step_factor
vymax := client::yStep * step_factor

numstepsx := |dx / vxmax|
numstepsy := |dy / vymax|
```

```
IF numstepsx > numstepsy THEN
        numsteps := numstepsx
        mover::b_xAxis := 1
        d0 := dx
        d1 := dy
        s := client::yStep
ELSE
        numsteps := numstepsy
        mover::b_xAxis := 0
        d1 := dx
        d0 := dy
        s := client::xStep
FI

mover::dx := dx / numsteps
mover::dy := dy / numsteps

mover::b_di := - |d0|
mover::b_i1 := 2 * (|d1| - |s * numsteps|) * |d0|
mover::b_incr := d1 / |d1|
mover::b_i2 := mover::b_d1 * 2
```

5.5.2.98 Kernel function 0x54: DoBresen()

kfunct 0x55: DoBresen();
; Returns: (void)

Executes the Bresenham algorithm on the values calculated by InitBresen, and counts down the number of steps. It then invokes CanBeHere() on the resulting coordinates, and sets the new coordinates if it actually Can Be There.

5.5.2.99 Kernel function 0x55: DoAvoider(HeapPtr)

kfunct 0x55: DoAvoider();
HeapPtr avoider; Returns: (word) New direction

This function is a no-op in later SCI games, but is implemented in some or all pre-0.000.576 interpreters.

5.5.2.100 Kernel function 0x56: SetJump(?)

5.5.2.101 Kernel function 0x57: SetDebug()

kfunct 0x57: SetDebug();
; Returns: (void)

This function forces the interpreter to enter debug mode. It is equivalent to pressing LShift-RShift-PadMinus.

5.5.2.102 Kernel function 0x58: InspectObj(?)

5.5.2.103 Kernel function 0x59: ShowSends(?)

5.5.2.104 Kernel function 0x5a: ShowObjs(?)

5.5.2.105 Kernel function 0x5b: ShowFree(?)

5.5.2.106 Kernel function 0x5c: MemoryInfo(word)

kfunct 0x5c: word mode();
word mode;
(word) mode: 0 to 4 (see below)

Returns: (word) The amount of free memory on the heap, in bytes

This function returns the total amount of free memory on the heap if mode == 0. If mode equals 1, the total size of the largest chunk of heap memory is returned. In mode 2, the size of the largest available hunk memory block is returned, and mode 3 returns the total amount of free hunk memory, shiftet to the right by 4 bits.

Mode 4 was apparently introduced in SCI01 and reports the amount of free memory provided by DOS in paragraphs.

5.5.2.107 Kernel function 0x5d: StackUsage(?)

5.5.2.108 Kernel function 0x5e: Profiler(?)

5.5.2.109 Kernel function 0x5f: GetMenu(word, word)

Parameters:

entry	: **word**	A pair of bytes. In LE notation, the higher byte is the "menu ID", and the lower byte is the "entry ID".
key	: **word**	A special key selecting some particular information regarding the menu entry.

Retrieves some metainformation about an (existing) menu entry. *entry* selects the menu and entry the information is recovered with respect to, and *key* specifies which particular information to recover. At the moment, the following kinds of information are known for *key*:

ID	FreeSCI macro (`MENU_ATTRIBUTE_...`)	Description
0x6d	`SAID`	The "*Said*" spec associated with the menu entry, or a null pointer. If this spec is matched, the next `GetEvent()` call will behave as if the appropriate menu option had been selected.
0x6e	`TEXT`	The string currently displayed for the menu item.
0x6f	`KEY`	An optional key (as reported by `GetKey()`) the menu option should be triggered by.
0x70	`ENABLED`	Whether the menu option is enabled or not (in the latter case, it is grayed out and cannot be selected).
0x71	`TAG`	A value without special semantics.

5.5.2.110 Kernel function 0x60: SetMenu(word, [word, any]*)

Parameters:

entry	: **word**	A pair of bytes. In LE notation, the higher byte is the "menu ID", and the lower byte is the "entry ID".
key	: **word**	A special key selecting some particular information regarding the menu entry.
value	: **word**	A special key selecting some particular information regarding the menu entry.

`SetMenu` is a varargs function; it takes a menu bar *entry* ID (cf. `GetMenu`, section 5.5.2.109) followed by any even number of parameters. Each of these parameter pairs begins with a *key*; the second entry is a *value*, whose type depends on the key. Semantics of *key* are as in `GetMenu` (cf. section 5.5.2.109).

5.5.2.111 Kernel function 0x61: GetSaveFiles(String, String, HeapPtr*)

kfunct 0x61: GetSaveFiles();
String game_id, String strspace, HeapPtr *ptrs;

(String) game_id: The game ID as a string

(String) strspace: The string which the result should be stored in

(HeapPtr *) ptrs: The array of pointers which the string pointers are to be stored in

Returns: (word) The number of savegames for the specified game_id.

Returns an array of strings describing the existing save games for game_id. The strings are put into strspace one by one, and heap pointers to each of them are put into the ptrs array. The number of saved games is returned in the accumulator.

5.5.2.112 Kernel function 0x62: GetCWD(HeapPtr)

kfunct 0x62: GetCWD();
HeapPtr address;
(HeapPtr) address: The address to write to
Returns: (HeapPtr) The supplied address
This function retreives the current working directory (CWD) and stores its string representation at the location pointed to by the supplied parameter.
FreeSCI returns a sub-directory of the user's home directory, if applicable, instead of the cwd.

5.5.2.113 Kernel function 0x63: CheckFreeSpace(String)

kfunct 0x63: CheckFreeSpace();
String path;
(String) path: The path to examine
Returns: (word) 1 if saving is possible, 0 otherwise
Returns TRUE if there would be enough space left on the specified path to save the current game (but doesn't actually save).

5.5.2.114 Kernel function 0x64: ValidPath(?)

5.5.2.115 Kernel function 0x65: CoordPri(?)

5.5.2.116 Kernel function 0x66: StrAt (String, word[, char])

kfunct 0x66: StrAt();
String src, word offset[, char replacement];
(String) src: The string to read from
(word) offset: The offset inside the string
(char) replacement: An optional replacement value for the indexed character
Returns: (char) The character requested
This function retreives a single character from a string. Optionally, if `replacement` is set, the source character will be replaced with the specified `replacement`.

5.5.2.117 Kernel function 0x67: DeviceInfo(word, String[, String])

kfunct 0x67: DeviceInfo();
word sub_function, String string1[, String string2];
(word) sub_function: A numeric value from 0 to 3, inclusive. See below.
(String) string1: See below.
(String) string2: See below.
Returns: See below
Depending on the value of sub_function, this system call executes one of four defined actions:
0 GET_DEVICE
1 GET_CURRENT_DEVICE
2 PATHS_EQUAL
3 IS_FLOPPY
See the specific function definitions below for more information.

5.5.2.118 Kernel function 0x67: DeviceInfo(GET_DEVICE, String, String)

kfunct 0x67: DeviceInfo();
word GET_DEVICE, String input, String output;
(word) GET_DEVICE: Constant sub-function identifier (0)
(String) input: A path whose device identifier should be extracted
(String) output: The destination of the device identifier
Returns: (HeapPtr) Points to the terminating zero character of output
GET_DEVICE returns the drive/device on which "input" resides in output (and a pointer to the terminating NULL in the accumulator).

5.5.2.119 Kernel function 0x67: DeviceInfo(GET_CURRENT_DEVICE, String output)

kfunct 0x67: DeviceInfo();
word GET_CURRENT_DEVICE, String output;
(word) GET_CURRENT_DEVICE: Constant sub-function identifier (1)
(String) output: The destination which the CWD device identifier should be written to.
Returns: (HeapPtr) Points to the terminating null character of output
GET_CURRENT_DEVICE returns the drive/device that contains the current working directory (and a pointer to the terminating NULL in the accumulator)

5.5.2.120 Kernel function 0x67: DeviceInfo(PATHS_EQUAL, String path1, String path2)

kfunct 0x67: DeviceInfo();
word PATHS_EQUAL, String path1, String path2;
(word) PATHS_EQUAL: Constant sub-function identifier (2)
(String) path1: First path to compare
(String) path2: Second path to compare
Returns: (word) 1 if path1 and path2 point to the same physical location, 0 otherwise.
PATHS_EQUAL returns TRUE if the two supplied paths point to the same place.

5.5.2.121 Kernel function 0x67: DeviceInfo(IS_FLOPPY, String path)

kfunct 0x67: DeviceInfo();
word IS_FLOPPY, String path;
(word) IS_FLOPPY: Constant sub-function identifier (3)
(String) path:
Returns: (word) 1 if *path* is on a floppy disk, 0 otherwise
PATHS_EQUAL returns TRUE if the two supplied paths point to the same place.

5.5.2.122 Kernel function 0x68: GetSaveDir()

kfunct 0x68: GetSaveDir();
; Returns: (String)
This function returns the heap position allocated to store the string representation of the save game directory. This heap space is allocated automatically during startup.

5.5.2.123 Kernel function 0x69: CheckSaveGame(String, word[, String])

kfunct 0x69: CheckSaveGame();
String game_id, word game_nr[, String version];
(String) game_id: The savegame ID string
(word) game_nr: The savegame number
(String) version: An optional game version string
Returns: (word) 1 if the savegame is loadable, 0 otherwise
Returns TRUE if the specified save game is valid and loadable (i.e., not for another game/interpreter/version).

5.5.2.124 Kernel function 0x6a: ShakeScreen(word[, word])

kfunct 0x6a: ShakeScreen();
word times [, word direction];
(word) *times*: Number of times to shake the screen
(word) *direction*: See below
Returns: (void)
If *direction* is not specified, it defaults to 1. It is a bitmask and defined as follows:
bit 0 Shake 10 pixels downwards
bit 1 Shake to the right
bit 2 Unknown, but used

5.5.2.125 Kernel function 0x6b: FlushResources(?)

5.5.2.126 Kernel function 0x6c: SinMult(?)

5.5.2.127 Kernel function 0x6d: CosMult(?)

5.5.2.128 Kernel function 0x6e: SinDiv(?)

5.5.2.129 Kernel function 0x6f: CosDiv(?)

5.5.2.130 Kernel function 0x70: Graph(?)

5.5.2.131 Kernel function 0x71: Joystick(word, word)

kfunct 0x71: Joystick();
word subfunction, word param;
(word) subfunction: Always 0x0c
(word) param: Parameter for the subfunction, purpose unknown.
Returns: (void)

Chapter 6

SCI in action

6.1 Event handling in SCI

By Lars Skovlund

Version 1.0, 12. July 1999

This article will deal with the event manager in SCI. Like several other key parts of the interpreter, this one actively communicates with the SCI application. It directly writes to objects of the Event class, but more on that later.

The different input devices are polled differently:

- The keyboard is typically polled at each timer tick (which is 60 hz).
- SCI sets up a callback for the PC mouse driver, meaning that the mouse driver "polls itself" and sends information to the interpreter. On non-MS-DOS platforms, this would probably be done in the timer handler. [1]
- The joystick is only polled when the script wants to.

Some parts of the event mechanism (in particular, keyboard management) are very PC specific, and a conversion will no doubt have to take place on other platforms.

6.1.1 Event types and modifiers

There are three types of events, distinguished by their "type" property. The possible values are listed below; they are laid out as a bitfield to allow for selective event retrieval, see later.

0x00 Null event
0x01 Mouse button event
0x02 Mouse button release event
0x04 Keyboard event
0x40 Movement (joystick) event

This type is returned to the SCI event managers by the input device drivers along with a "message" and a set of "modifiers". This is the basic event structure, although some event types contain extra information. The latter field is a direct copy of the BIOS shift flags, laid out as follows:

bit 7 Insert active
bit 6 Caps lock active
bit 5 Num lock active
bit 4 Scroll lock active
bit 3 Alt key pressed
bit 2 Ctrl key pressed
bit 1 Left shift key pressed
bit 0 Right shift key pressed

It is obvious, then, that these keys by themselves don't generate any keyboard events. They can, however, be combined with other keys or mouse clicks to produce "shift-click" events, for instance.

[1] The default FreeSCI event mechanism uses libgii, which is completely event-based.

6.1.1.1 The null events

These are generated when a script wants to see an event, but there isn't one to give. The current tick count and mouse position. The tick count, as explained in another document, is the time passed since the interpreter started, measured in 1/60ths of a second. It doesn't seem to be copied into the event object, however.

6.1.1.2 The mouse events

The mouse position is returned in extra fields in the event record.

If the middle or right button is pressed, this is reflected by the modifiers, in addition to the mouse event. The middle button is translated to the Ctrl key (i.e. set modifiers bit 2), the right button "holds down" both shift keys (setting bits 1 and 0). Every SCI interpreter (at least from 0.000.572 and up) does this, but to my knowledge it is used only in QfG2, where either a shift-click or a right-click is equivalent to typing "look ...".

6.1.1.3 The keyboard event

The keyboard driver also generates events. When a key is pressed, a keyboard event is generated, with the message field set to the scan code of the pressed key. It should be simple enough, right? Not quite so. The script may want to know if a direction key was pressed, and if so, which. It may call the KMapKeyToDir kernel function for this. KMapKeyToDir takes a keyboard event as input and converts it to a movement event, which is described next.

6.1.1.4 The movement event

The movement event is only generated by the joystick driver. However, on request, the keyboard driver can convert keyboard events into movement events as described above. The message field is just a direction code, mapped as follows:

8	1	2
7	Center	3
6	5	4

That is, the direction code starts at straight up (code 1), increasing with clockwise movement.

6.2 The Parser

6.2.1 Vocabulary file formats

By Lars Skovlund
Version 1.0, 30. July 1999

6.2.1.1 The main vocabulary (VOCAB.000)

The file begins with a list of 26 offsets. Each index corresponds to a letter in the (English) alphabet, and points to the first word starting with that letter. The offset is set to 0 if no words start with that letter. If an input word starts with an alphabetical letter, this table is used to speed up the vocabulary searching - though not strictly necessary, this speeds up the lookup process somewhat.

After the offset table are the actual words. A word defition consists of two parts: The actual text of the word, compressed in a special way, and a 24-bit (yes, three bytes) ID. The ID divided in 2 12-bit quantities, a word class (grammatically speaking) mask, and a group number. The class mask is used, among other things, for throwing away unnecessary words. "Take book", for instance, is a valid sentence in parser'ese, while it isn't in English.

The possible values are arranged as a bit field to allow for class masks, see later. Only one bit is actually tested by the interpreter. If a word class equals to 0xff ("anyword"), the word is excluded (allowing for parser'ese). The values go like this:

0x001 number (not found in the vocabulary, set internally)
0x002 special
0x004 special
0x008 special[2]
0x010 preposition
0x020 article
0x040 qualifying adjective
0x080 relative pronoun
0x100 noun
0x200 indicative verb (such as "is", "went" as opposed to _do_ this or that, which is imperative)
0x400 adverb
0x800 imperative verb

The group number is used to implement synonyms (words with the same meaning), as well as by the Said instruction to identify words. There is also a way of using synonyms in code, see the appropriate document.

The compression works in this way: Each string starts with a byte-sized copy count. This many characters are retained from the previous string. The actual text comes after, in normal low-ascii. The last character in the text has its high bit set (no null termination!).

Here is an example of the compression scheme:

apple 0,appl\0xE5

The byte count is 0 because we assume that "apple" is the first word beginning with an a (not likely, though!). 0xE5 is 0x65 (the ascii value for 'e') | 0x80. Watch now the next word:

athlete 1,thlet\0xE5

Here, the initial letter is identical to that of its predecessor, so the copy count is 1. Another example:

atrocious 2,rociou\0xF3

6.2.1.2 The suffix vocabulary (VOCAB.901)

For the following section, a reference to a grammar book may be advisable.

The suffix vocabulary is structurally much simpler. It consists of variably-sized records with this layout:

NULL-TERMINATED	Suffix string
WORD	The class mask for the suffix
NULL-TERMINATED	Reduced string
WORD	The output word class

The suffix vocabulary is used by the interpreter in order to parse compound words, and other words which consist of more than one part. For instance, a simple plural noun like "enemies" is reduced to its singular form "enemy", "stunning" is converted to "stun" etc. The point is that the interpreter gets a second chance at figuring out the meaning if the word can not be identified as entered. A word which changes its class does might end up as a different word class, the correct class is always retained. Thus, "carefully", an adverb, is reduced to its adjectival form "careful", and found in the vocabulary as such, but it is still marked as an adverb.

The suffix vocabulary consists of variably-sized records with this layout:

NULL-TERMINATED	Suffix string
WORD	The output word class
NULL-TERMINATED	Reduced string
WORD	The allowed class mask for the reduced word

An asterisk (*) represents the word stem. Taking the above example with "enemies", the interpreter finds this record:

```
*ies
```
0x100
```
*y
```
0x100

word class 0x100 being a noun.

The interpreter then tries to replace "enemies" with "enemy" and finds that word in the vocabulary. "Enemy" is a noun (class 1), which it is also supposed to be, according to the suffix vocabulary. Since we succeeded, the word class is set to the output value (which is, incidentally, also 1).

Numbers If the word turns out to be a number (written with numbers, that is), and that number is not listed explicitly in the vocabulary, it gets an ID of 0xFFD, and a word class of 0x100.

6.2.1.3 The tree vocabulary (VOCAB.900)

This vocabulary is used solely for building parse trees. It consists of a series of word values which end up in the data nodes on the tree. It doesn't make much sense without the original parsing code.

6.2.2 The black box: The magic behind Sierra's text parser

By Lars Skovlund

Version 0.1, 30. July 1999. Incomplete!

This document describes the process of parsing user input and relating it to game actions. This document does not describe the process of the user typing his command; only the "behind-the-scenes" work is described, hence the title.

The process of parsing is two-fold, mainly for speed reasons. The Parse kernel function takes the actual input string and generates a special "said" event (type 0x80) from it. This function is only called once per line. Parse can either accept or reject the input.

A rejection can only occur if Parse fails to identify a word in the sentence.

Even if Parse accepts the sentence, it does not need to make sense. Still, syntax checks are made - see later.

Assuming that the parsing succeeded, the User object (which encapsulates the parser) then goes on to call the relevant event handlers. These event hand- lerrs in turn call the Said kernel function. This function is potentially called hundreds or even thousands of times, so it must execute as quickly as possible. Said simply determines from the pre-parsed input line whether or not a specific command is desired.

The Parse function must always work on an internal copy of the actual string, because the user must be able to recall his exact last input using the F3 key. The parser's first step is to convert the input line to pure lower case. This is because the vocabulary words are entered in lower case. The parser then searches the main vocabulary (VOCAB.000), hoping to find the word.

This doesn't necessarily happen yet. Consider, for example, the meaning of the word "carefully", which does not appear in the vocabulary, but is found anyway. This is due to the so-called suffix vocabulary, which is discussed in another document.

If the word still can't be found, the interpreter copies the failing word into a buffer temporarily allocated on the heap (remember, the interpreter operates on its own local buffers). It then calls the Game::wordFail method which prints an appropriate message. The interpreter then deallocates the buffer and exits (it does, however, still return an event. The claimed property of that event is set to TRUE to indicate that the event has already been responded to (error message printed)).

If the interpreter succeeds in identifying all the words, it then goes on to check the syntax of the sentence - it builds a parse tree. See the appropri- ate document.

If the syntax of the sentence is invalid, the interpreter calls Game::syntaxFail, passing the entire input line. As for the error situation, the event is claimed.

As mentioned in the beginning of this text, this function generates an event. This event, apart from its type id, does not contain any data. Rather, all pertinent data is kept in the interpreter.

The Said kernel function is called for each command which the game might respond to at any given time. Its only parameter is a pointer to a said information block which resides in script space. This block is described below (see Section 6.2.4).

The Said function first does some sanity checking on the event pointer which Parse stored earlier. It must be a said event (type property), and it must not have been handled by an earlier call to Said (claimed property).

It then word-extends the passed said block into a temporary buffer (command codes are byte-sized, remember?). This is supposedly just for convenience/speed, and not really needed.

6.2.3 The Parse tree

This and the two following sections borrow some ideas and structures from abstract language theory. Readers might want to consider related literature.

Most of the information explained here was gathered by Lars Skovlund, and, before that, Dark Minister.

After tokenizing, looking up, and finally aliasing the data found in the parsed input string, the interpreter proceeds to build a parse tree T_Π from the input tokens

$$I := w_0, w_1, w_2 \dots w_{n-1}$$

where

- $w_j \in W$
- $\gamma_j \in \Gamma$
- $\mu_j \in 2^C$
- $w_j = (\gamma_j, \mu_j$

with W being the set of all words, Γ being the set of all word groups, C being the set of all class masks $\{1, 2, 4, 8, 10, 20, 40, 80, 100\}$, γ_j being the word group w_j belongs to, and μ_j being its class mask, as described above.

For the following sections, we define

- group : $W \to \Gamma$.group : $(\gamma, \mu) \mapsto \gamma$
- classes : $W \to C$.classes : $(\gamma, \mu) \mapsto \mu$
- $C_x = \{w | w \in W.x \in \text{class}(w)\}$

To do that, it uses the class masks M as input for a pushdown automaton (PDA) A built from a parser grammar; if M was accepted by A, the parse tree T_Π will be built from the matching syntax tree to represent the semantics.

The PDA is defined by a grammar $G = (V, \Sigma, P, s)$, most of which, along with its semantics, is stored in `vocab.900`. This resource contains a parser rule at every 20 bytes, starting with a non-terminal symbol v (one word) and a null-terminated list of up to five tuples $\langle \sigma_i, m_i \rangle$, both of which are words. In these tuples, m_i is a terminal or non-terminal symbol (determined by σ_i), and σ_i is the meaning of m_i:

σ_i	Type	Meaning
0x141	Non-terminal	Predicate part: This identifies the first part of a sentence
0x142	Non-terminal	Subject part: This identifies the second part of a sentence
0x143	Non-terminal	Suffix part: This identifies the third and last part of a sentence
0x144	Non-terminal	Reference part: This identifies words that reference another word in the same sentence part
0x146	Terminal	Match on class mask: Matches if $m_i \in \text{classes}(w_j)$
0x14d	Terminal	Match on word group: Matches if $(m_i = \text{group}(w_j)$
0x154	Terminal	”Force storage”: Apparently, this was only used for debugging.

With the notable exception of the first rule, these rules constitute P. $V := \{x | \exists R \in P.x \in R\}$; typically, $V = \{0x12f ... 0x13f\}$. $s = m_0$ of the first rule encountered; in all games observed, it was set to 0x13c. Σ contains all word groups and class masks. For the sake of simplicity, we will consider rules matching composite class masks to be several rules. Here is a simplified example of what such a grammar might look like (the hexadecimal prefix '0x' is omitted for brevity):

In addition to this grammar, each right-hand non-terminal m_i carries its semantic value ρ_i, which is not relevant for constructing a syntax tree, but must be considered for the semantic tree T_Π. These values were omitted in the example above. As in the example above, the grammar is a context-free (type 2) grammar, almost in Chomsky Normal Form (CNF) in SCI; constructing a grammar with CNF rules from it would be trivial. [3]

Obviously, G is an ambiguous grammar. In SCI, rule precedence is implied by rule order, so the resulting left derivation tree is well-defined (in the example, it would be defined by D_0) [4].

[3] FreeSCI constructs a GNF (Greibach Normal Form) representation from these rules for parsing.

[4] In FreeSCI, you can use the ”parse” console command to retreive all possible left derivation trees

Exemple 6.2.1 Parse grammar example

$$
\begin{aligned}
G &= \langle \{\rangle 12f ... 13e\}, \{C_1, C_2, C_4, \ldots, C_100\}, P, 13c) \\
P &= \{ \begin{array}{lcl} 13c & \rightarrow & 13b\ 134 \\ 13c & \rightarrow & 13b\ 13d\ 133 \\ 13c & \rightarrow & 13b\ 13d \\ 13c & \rightarrow & 13b \\ 13c & \rightarrow & 13b\ 13d\ 13b\ 13d \\ 13b & \rightarrow & 131\ 134 \\ 13b & \rightarrow & 131\ 13d\ 13d \\ 13b & \rightarrow & 131 \\ 13d & \rightarrow & 134 \\ 131 & \rightarrow & C_8 0 \\ 133 & \rightarrow & C_2 0 \\ 134 & \rightarrow & C_1 0 \} \end{array}
\end{aligned}
$$

6.2.3.1 Semantics

This is important, since the parser does much more than just accept or discard input. Using the semantic tags applied to each non-terminal on the right-hand side of a rule, it constructs what I will call the semantic parse tree T_Π, which attempts to describe what the input means. For each non-terminal rule

$$r = v_0 \rightarrow v_1 v_2 \ldots v_n$$

there are semantic tags $\sigma_{r,1}, \sigma_{r,2} \ldots \sigma_{r,n} \in S$, as explained above. T_Π is now constructed from the resulting derivation and the semantic tags assiociated with each non-terminal of the rule used. The construction algorithm is explained below with T_Π being constructed from nodes, which have the following structure:

$$\text{NODE} = \{\diamond\} \cup S \times V \times (\text{NODE} \cup \Gamma)*;$$

Where S is the set of possible semantic values, and V is the set of non-terminals as defined in the grammar. We will also use the sequence $\gamma_0, \gamma_1, \gamma_2 \ldots \gamma_{k-1}$, which will represent the word groups the input tokens belonged to (in the exact order they were accepted), and the sequence $r_0, r_1, r_2 \ldots r_{l-1}$, which will be the list of rules used to create the left derivation tree as described in the previous section.

```
Helper function sci_said_recursive: S \times V \times (V \cup \Sigma)* \
Parameters: s \in S, Rule r \in V \times (V \cup \Sigma): v0 \to v1 v2
cnmr = cnr
\Node n := s, v0
FOR j := 1 TO i
          IF (vj \in \Sigma) THEN
                  n := n, \gammacn\gamma
                  cn\gamma := cn\gamma + 1
          ELSE
                  cnoldr := cnr
                  cnr := cnr + 1
                  n := n, sci_said_recursive(\sigmarmr,j, rcnoldr)
          FI
ROF
RETURN (n)


Helper function get_children: \Node \to \Node*
                get_children((s, v, n0, n1 ... nm)) := n0, n1 ... nm


Algorithm SCI-SAID-TREE
```

Exemple 6.2.2 Parser example

Parse is called with "`open door`".

- "`open`" $\in \langle 842, \{C_8 0\}\rangle$ (an imperative word of the word group 0x842)
- "`door`" $\in \langle 917, \{C_1 0\}\rangle$ (a substantive of the word group 0x917)
- $I = \langle 842, \{C_8 0\}\rangle, \langle 917, \{C_1 0\}\rangle$

I is clearly accepted by automatons based on the grammar described above. Here are two possible derivations:

$$\begin{aligned} & D_0 = && 13c \\ (13c \to 13b134) & && \vdash 13b\ 134 \\ (13b \to 131) & && \vdash 131\ 134 \\ (131 \to C_{80}) & && \vdash C_{80}\ 134 \\ (134 \to C_{10}) & && \vdash C_{80}\ C_{10} \end{aligned}$$

$$\begin{aligned} & D_1 = && 13c \\ (13c \to 13b) & && \vdash 13b \\ (13b \to 131134) & && \vdash 131\ 134 \\ (131 \to C_{80}) & && \vdash C_{80}\ 134 \\ (134 \to C_{10}) & && \vdash C_{80}\ C_{10} \end{aligned}$$

```
cn\gamma := 0;
cnr := 1;
ntemp := ntemp, SCI-SAID-RECURSIVE(0, r0)
root(T\Pi) := (141, 13f, get_children(ntemp))
```

Here is an example, based on the previous one:

6.2.4 Said specs

To test what the player wanted to say, SCI compares T_Π with a second tree, T_Σ, which is built from a so-called Said spec. A Said spec is a variable-sized block in SCI memory which consists of a set of byte-sized operators and special tokens (stored in the range 0xf0 to 0xf9) and word groups (in big-endian notation, so that they don't conflict with the operators); it is terminated by the special token 0xff. The meanings of the operators and special tokens are as follows:

Exemple 6.2.3 Semantic tree example

- $k = 2$
- $\gamma_0 = 842$
- $\gamma_1 = 917$
- $l = 4$
- $r_0 = 13c \to 13b\ 134$
- $\sigma_{r_0,1} = 141$
- $\sigma_{r_0,2} = 142$
- $r_1 = 13b \to 131$
- $\sigma_{r_1,1} = 141$
- $r_2 = 131 \to C_{80}$
- $r_3 = 134 \to C_{10}$

The resulting tree would look like this:

```
(141 13f
        (141 13b
                (141 131 842)
        )
        (142 134 917)
)
```

Operator	Byte representation	Meaning
,	f0	”OR”. Used to specify alternatives to words, such as ”take , get”.
&	f1	Unknown. Probably used for debugging.
/	f2	Sentence part separator. Only two of these tokens may be used, since sentences are split into a maximum of three parts.
(	f3	Used together with ’)’ for grouping
)	f4	See ’(’
[	f5	Used together with ’[’ for optional grouping. ”[*a*]” means ”either *a* or nothing”
]	f6	See ’[’.
#	f7	Unknown. Assumed to have been used exclusively for debugging, if at all.
<	f8	Semantic reference operator (as in ”get < up”).
>	f9	Instructs Said() not to claim the event passed to the previous Parse() call on a match. Used for successive matching.

This sequence of operators and word groups is now used to build the Said tree T_Σ. I will describe the algorithm used to generate T_Σ by providing a grammar G_Σ, with $L(G_\Sigma)$ containing all valid Said specs. The semantics will be provided under each rule with a double arrow:

```
G\Sigma = ({saidspec, optcont, leftspec, midspec, rightspec, word, cwordset, w

P := {
  saidspec \to   leftspec optcont
                        \Rightarrow (141 13f leftspec optcont)
                | leftspec midspec optcont
                        \Rightarrow (141 13f leftspec midspec optcont)
                | leftspec midspec rightspec optcont
                        \Rightarrow (141 13f leftspec midspec rightspec optcon



  optcont \to   e
                        \Rightarrow
                | >
                        \Rightarrow (14b f900 f900)



  leftspec \to  e
                        \Rightarrow
                | expr
                        \Rightarrow (141 149 expr)
```

```
midspec \to    / expr
                      \Rightarrow (142 14a expr)
               | [ / expr ]
                      \Rightarrow (152 142 (142 14a expr))
               | /
                      \Rightarrow


rightspec \to  / expr
                      \Rightarrow (143 14a expr)
               | [ / expr ]
                      \Rightarrow (152 143 (143 14a expr))
               | /
                      \Rightarrow

word \to       \gamma \in \Gamma
                      \Rightarrow (141 153 \gamma)


cwordset \to   wordset
                      \Rightarrow (141 14f wordset)
               | [ wordset ]
                      \Rightarrow (141 14f (152 14c (141 14f wordset)))


wordset \to    word
                      \Rightarrow word
               | ( expr )
                      \Rightarrow (141 14c expr)
               | wordset , wordset
                      \Rightarrow wordset wordset
               | wordset , [ wordset ]
                      \Rightarrow wordset wordset


expr \to               cwordset cwordrefset
                      \Rightarrow cwordset cwordrefset
               | cwordset
                      \Rightarrow cwordset
               | cwordrefset
                      \Rightarrow cwordrefset


cwordrefset \to        wordrefset
                      \Rightarrow wordrefset
               | [ wordrefset ]
                      \Rightarrow (152 144 wordrefset)


wordrefset \to        < wordset recref
                      \Rightarrow (144 14f word) recref
               | < wordset
                      \Rightarrow (144 14f word)
               | < [ wordset ]
                      \Rightarrow (152 144 (144 14f wordset))
```

```
  recref \to    < wordset recref
                        \Rightarrow (141 144 (144 14f wordset) recref)
                | < wordset
                        \Rightarrow (141 144 (144 14f wordset))
}
```

6.2.5 Matching the trees

The exact algorithm used to compare T_Π to T_Σ is not known yet. The one described here is based on the approximation used in FreeSCI, which is very similar to the original SCI one.

First, we need to describe a set of functions for traversing the nodes of T_Σ and T_Π, and doing some work. They will be operating on the sets $\mathbb{N}$ (all non-negative integral numbers), $\mathbb{B} = \{\text{tt}, \text{ff}\}$ (Booleans), and NODE (which we defined earlier).

$$\begin{aligned}
\text{first} &: \text{NODE} \to S \\
\text{first} &: \langle s, v, n_0, n_1 \dots n_i \rangle \mapsto s \\
\\
\text{second} &: Node \to V \\
\text{second} &: \langle s, v, n_0, n_1 \dots n_i \rangle \mapsto v \\
\\
\text{word} &: Node \to \Gamma \\
\text{word} &: \langle s, v, \gamma \rangle \mapsto \gamma \\
\\
\text{children} &: \text{NODE} \to \text{NODE}* \\
\text{children} &: \langle s, v, n_0, n_1 \dots n_i \rangle \mapsto \{m | \forall m.m \in \{n_0, n_1 \dots n_i\} \wedge m \in Node\} \\
\\
\text{all_children} &: \text{NODE} \to \text{NODE}* \\
\text{all_children} &: n \mapsto \text{children}(n) \cup \{m | \exists l.l \in \text{all_children}(n).m \in l\} \\
\\
\text{is_word} &: \text{NODE} \to B \\
\text{is_word} &: \langle s, v, n_0, n_1 \dots n_i \rangle = \text{tt} \iff (i = 0) \wedge n_0 \in \Gamma \\
\text{verify_sentence_part_elements} &: \text{NODE} \times \text{NODE} \to B \\
\text{verify_sentence_part_elements} &: \langle n_p, n_s \rangle \mapsto \text{tt} \iff (\text{first}(n_s = 152) \wedge ((\forall n \\
\\
\text{verify_sentence_part} &: \text{NODE} \times \text{NODE} \to B \\
\text{verify_sentence_part} &: \langle n_p, n_s \rangle \mapsto \text{tt} \iff \forall n.n \in \text{children}(n_s) : \\
\\
\text{verify_sentence_part_brackets} &: \text{NODE} \times \text{NODE} \to B \\
\text{verify_sentence_part_brackets} &: \langle n_p, n_s \rangle \mapsto \text{tt} \iff (\text{first}(n_p) = 152 \wedge (\forall m
\end{aligned}$$

With these functions, we can now define an algorithm for augmenting T_Π and T_Σ:

Algorithm SCI-AUGMENT matched := ttclaim_on_match := tt `FOREACH` $n \in \text{root}(T_\Sigma)$ `IF` $((\text{first}(n) = 14b) \wedge (\text{second}(n) = f900))$ `THEN` claim_on_match := ff `ELSE IF` $\neg$verify_sentence_part_brackets$(n, \text{root}(T_\Pi))$ `THE` ff `END-FOREACH`

Augmenting succeeded if matched = tt; in this case, T_Π is one of the trees accepted by the description provided by T_Σ. In this case, Said() will return 1. It will also claim the event previously provided to Parse(), unless claim_on_match = ff.

6.3 Views and animation in SCI

by Lars Skovlund

Version 0.2, 4. January 2002, with notes by Christoph Reichenbach

This chapter deals with a rather complex subject within SCI. The subsystem described here is one of the ”bad boys” in SCI, since it calls functions in user space, as well as changing the value of various selectors. This document is not necessarily complete. There are several things I have not covered - because they are better off in a separate document, or simply because I haven't yet figured that part out. IOW, this stuff is incomplete. Things may change.

After drawing a pic on the screen (which is DrawPic's job, that doesn't surprise now, does it?), some views have to be added to it. There are two ways of doing this; the AddToPic and the Animate call. While AddToPic is used for static views, Animate lets each animated view in the cast list perform an ”animation cycle”.

An animation cycle is done entirely in SCI code (with the aid of some kernel calls). It involves two other objects; the mover and the cycler. The mover is responsible for controlling the motion of an actor towards a specific point, while the cycler changes the image of the actor, making him appear to walk, for instance.

The behaviour of a view is controlled by its signal property. This property contains a bitfield which describes a lot of animation-related stuff. The bits can be roughly divided into two groups; the script and interpreter bits (I had called them Option and State bits at first, but that is not entirely accurate). The first group allows the script to influence the drawing pro- cess somewhat, the other are used internally by the interpreter. The two groups overlap a bit, though.

The unlisted bits are probably all interpreter bits. They don't seem to have an effect when set. Many bits seem to be involved in the decision whether to display a view or not. I have not completely figured this out. [5]

Animate (see Section 5.5.2.12) can be called in two ways:

Animate(DblList cast, bool cycle)

Animate()

If the second syntax is used, the two parameters are assumed to be zero.

The cast list is just a list of the views to draw. Animate creates a backup of this list for updating purposes. However, this backup cast list isn't just a normal copy. The interpreter copies some selectors from the view (view, loop, cel, nsRect) and places them in a special data structure. This indicates to me that there is a possibility that the view objects may be deleted even though an update is anticipated.

The general pseudocode for Animate goes as follows:

```
0. Backup PicNotValid: PicNotValid' := PicNotValid
1. If we don't have a new cast:
    1.1. if PicNotValid is set:
        1.1.1. Redraw picture with opening animation
    1.2. exit
2. For each view in the cast list:
    2.1. If view is not frozen:
        2.1.1. call view::doit(), performing an animation cycle
3. Prepare a list of y coordinates by traversing the cast list
4. For each view in the cast list:
    4.1. If the view resource view::view has not been loaded yet:
        4.1.1. Load view.nr, where nr=view::view
5. For each view in the cast list:
    5.1. If view::loop is invalid, set view::loop := 0
    5.2. If view::cel is invalid, set view::cel := 0
6. Sort the cast list, first by y, then by z
7. For each view in the cast list: Update view::nsRect (SetNowSeen())
8. For each view in the cast list: Unless the views' priority is fixed, recalculate it
9. For each view in the cast list:
    9.1. If NO_UPDATE is set for the view:
        9.1.1. If the following holds:
```

[5] The bit names I have written come from some debug information I got from QfG2 - type ”suck blue frog” then Ctrl-W to save the cast list!

```
            9.1.1.1.     (VIEW_UPDATED || FORCE_UPDATE)
            9.1.1.2.  || (!(VIEW_UPDATED || FORCE_UPDATE) && !IS_HIDDEN && REMOVE_VIEW
            9.1.1.3.  || (!(VIEW_UPDATED || FORCE_UPDATE) && !IS_HIDDEN && !REMOVE_VIE
            9.1.1.4.  || (!(VIEW_UPDATED || FORCE_UPDATE) && IS_HIDDEN && ALWAYS_UPDAT
            9.1.1.5. then increase PicNotValid by one.
        9.1.2. Clear the STOP_UPDATE flag
    9.2. otherwise:
        9.2.1. If (STOP_UPDATE and !ALWAYS_UPDATE) or (!STOP_UPDATE and ALWAYS_UPDATE)
            9.2.1.1. Increase PicNotValid by one
        9.2.2. Clear the FORCE_UPDATE flag
10. If PicNotValid is now greater than zero, call the sub-algorithm described separate
11. For each view: If NO_UPDATE, IS_HIDDEN and ALWAYS_UPDATE are not set:
    11.1. [12] Save the area covered by the view's nsRect, store the handle in view::u
    11.2. [13] Draw the view object
    11.3. [14] If the view IS_HIDDEN, clear the REMOVE_VIEW bit (don't need to hide it
    11.4. [15] Insert the view into the backup cast list
16. If PicNotValid', our copy of the initial value of PicNotValid, is non-zero:
    16.1. Refresh entire screen with opening animation
    16.2. PicNotValid := 0
17. For each view in the cast list:
    17.1. [18] If the view was changed in step 10 and neither REMOVE_VIEW nor NO_UPDAT
        17.1.1. [19] Redraw the nsRect and lsRect areas
        17.1.2. [20] Copy the nsRect to the lsRect
        17.1.3. [21] If IS_HIDDEN, set REMOVE_VIEW as well
22. For each view in the reverse cast list:
    22.1. [23] If neither NO_UPDATE nor REMOVE_VIEW is set:
        22.1.1. Restore the underbits
        22.1.2. Clear the underbits
    22.2. [24] if DISPOSE_ME is set, call view::dispose to dispose it
```

With the sub-algorithm being:

```
1. For each view from the cast list:
    1.1. [2] If NO_UPDATE is set:
        1.1.1. [3] If REMOVE_VIEW is set:
            1.1.1.1. If PicNotValid is 0, restore the area covered by view::underBits
            1.1.1.2. Free view::underBits
        1.1.2. [4] Clear FORCE_UPDATE
        1.1.3. [5] If VIEW_UPDATED is set: Clear VIEW_UPDATED and NO_UPDATE
    1.2. otherwise (if NO_UPDATE is not set):
        1.2.1. Clear STOP_UPDATE
        1.2.2. Set NO_UPDATE
6. For each view from the cast list:
    6.1. [7] Draw the view
    6.2. [8] If ALWAYS_UPDATE, clear STOP_UPDATE, VIEW_UPDATED, NO_UPDATE, FORCE_UPDAT
    6.3. [9] Clip the nsRect against the boundaries of the "natural" priority band of
    6.4. [10] If IGNORE_ACTOR is clear, fill the area found in 6.3. with 0xf on the co
11. For each view from the view list:
    11.1. if NO_UPDATE is set:
        11.1.1. [12] If IS_HIDDEN, then set REMOVE_VIEW, otherwise:
            11.1.1.1. clear REMOVE_VIEW
            11.1.1.2. [13] Save the area covered by the nsRect in the underBits
14. For each view from the cast list:
    14.1. If NO_UPDATE is set and IS_HIDDEN is clear:
        14.1.1. [15] Draw the view
```

Note that the ReAnimate subfunction (0x0D) of the Graph kernel call redraws parts of the maps

using the cast list created by Animate, whereas the ShowBits call (0x0C) copies parts of the active map to the physical screen.

6.4 The message subsystem

The message subsystem developed out of a desire to lessen the amount of coordinative work between dialogue writers and programmers. The text resource of early SCI suffered from the limitation of using a tuple $\langle module, message\text{-}id \rangle$[6] as index into the text resources. Worse, text resources were often generated by using special syntax in the source code, thus making the ordering prone to change as the code was extended and reorganized.

In 1990, Sierra had released its first icon-driven game, King's Quest V. The icon-based approach required a new approach to event handling. Each clickable object would be represented as an SCI instance. When an object was clicked, a method the corresponding instance would be called with a parameter specifying which icon was used. The action linked to each icon was called a verb, and the method doVerb.

Later, when creating the message interface, it was natural to re-use this notion, and couple the module and verb with a unique number for each clickable object (since the instance addresses are unpredictable). Although later versions of the message system were more complicated, these are the essentials of its first iteration.

Soon introduced was the concept of stage directions. They are directions to the voice talents in CD-ROM games. The important bit here is that the stage directions *are still present* in the shipped message files, and the interpreter must know how to remove them. Any string in parentheses **not** containing any lower case characters or digits *including* any whitespace following it is considered to be stage directions and is stripped before the script sees it. Character escapes (either in the form of literal-character escapes, such as \(, or escapes using the ASCII value in hex, such as \30) were supported in some versions.

Many versions of the message resource also support longer comments, meant for writer/coder communication. It is unclear how to derive their offsets in the resource, though.

The writers soon realised that the indexing model presented above was too simplistic. Developments in the game plot were still not handled adequately by the system, and required programmer assistance. In addition, the response to each action had to fit in one message box. Therefore, Sierra's programmers added two fields to the indexing model, namely *condition* (sometimes known as *case*) and *sequence*. The condition signified the state of the noun with respect to the game plot, in a manner of speaking. The sequence number allowed writers to write more than one screenful of text. Also, a piece of satellite data was introduced, namely the *talker*. The game might use this to display the face of the speaker on screen. One talker value was reserved for narrated parts, which don't display a face (but this is game-specific, and really outside the domain of the interpreter).

Even later, recursion was added. Actually, two kinds of recursion, which it is necessary to distinguish, were added. One involved resource-internal recursion, in which the writer decides to re-use a part of another dialogue by including a reference to it. This type of recursion was limited, though; it was impossible to refer to other modules, and the reference always pointed to the first message in a sequence. The other kind of recursion was controlled by the script, and was useful for such things as cut-scenes. The two types of recursion could be mixed freely, which is why there are both message stacks and message stack stacks in Sierra SCI (no kidding! see the included error message file INTERP.ERR or SIERRA.ERR).

6.4.1 Ties to audio

Of course, the story does not end there. CD-ROM games contain audio, and having two addressing schemes would have been a mess. So naturally, the same scheme was used. However, this poses another problem: Individual resources are usually addressed by just a type and a number, not by a message tuple like the one we saw above. Sierra's solution was to add a new resource type, other resource files and maps beside the main one. The extra resource files are called something like `RESOURCE.AUD` and `RESOURCE.SFX`, and their maps are contained in map resources, either in the main resource file or separately as patches. The resource type is called audio36, and there is a sync36 type as well which provides cueing capabilities to these resources (like sync does to ordinary audio resources, see section ???).

[6] While *module* was traditionally called *room* or *resource number*, I have chosen to use the terminology of the later message interface here.

Figure 6.1 SCI message resources and their capabilities

Ver.	StD	Cond/Seq	Rec	SRec
early[a]				
2.101	√			
3.340	√	√	√	
3.411	√	√	√	
4.000	√	√	√	
4.010	√	√	√	
4.211	√	√	√	
4.321	√	√	√	√
5.000	√	√	√	√

[a] May not include a version number (needs confirmation)

The maps are indexed by module (room number), so that 100.map contains map entries for all the message tuples that have the module number 100. Map number 65535 ($2^{16} - 1$) is special – it indexes ordinary audio resources.

To patch resources of this kind, Sierra used a base-36 encoding of the message tuple as the file name. Since this results in oddly looking names, the patch files, if any, are usually stored in a separate directory on the CD.

6.4.2 File formats

As can be seen from figure 6.1, the message file format and interfaces changed quite a bit over time. Interestingly, as perhaps the only part of the SCI system, message resource files incorporated a version number, with one exception. It is marked 'early' in the table. It is still possible to discern them from a corrupt resource, though.

The version numbers given in the table were divided by 1000 to yield an real-numbered representation; thus, the message format represented as 2.101 had a version tag of `0x835`.

All versions can be said to follow the general pattern given below; on the following pages, specific file formats are given for each version.

HEADER
MESSAGE OFFSETS
ACTUAL TEXT
COMMENTS/DEBUG

6.4.3 early

The exact file format is still not known, but seems to be the same as 2.101, without either the version number or the zero (Drantin?).

6.4.4 Version 2.101

The message resource begins with a 6-byte header, laid out thus:

Offset	Size (bytes)	Description
0	2	Version number (== 0x835)
2	2	Always zero
4	2	Number of messages in file (n)

The n offset records are laid out as follows:

Offset	Size (bytes)	Description
0	1	Noun
1	1	Verb
2	2	Offset to text (from beginning of resource)

6.4.5 Version 3.411

The message resource begins with an 8-byte header, laid out thus:

Offset	Size (bytes)	Description
0	2	Version number (== 0xd53)
2	2	Always zero
4	2	Pointer to first byte past text data, not counting this header
6	2	Number of messages in file (n)

The n offset records are laid out as follows:

Offset	Size (bytes)	Description
0	1	Noun
1	1	Verb
2	1	Condition
3	1	Sequence
4	1	Talker
5	2	Offset to text (from beginning of resource)
7	3	Unknown

6.4.6 Version 4.010

The message resource begins with an 8-byte header, laid out thus:

Offset	Size (bytes)	Description
0	2	Version number (== 0xfaa)
2	2	Always zero
4	2	Pointer to first byte past text data, not counting this header
6	2	Number of messages in file (n)

The n offset records are laid out as follows:

Offset	Size (bytes)	Description
0	1	Noun
1	1	Verb
2	1	Condition
3	1	Sequence
4	1	Talker
5	2	Offset to text (from beginning of resource)
7	1	Noun of referenced message
8	1	Verb of referenced message
9	1	Condition of referenced message

The reference fields are set to zero if no reference is intended.

6.5 Notes on the kernel calls

Two versions of this API were used, the early GetMessage() which had only one function, and a later subfunction-based one. While GetMessage() required one to specify noun/module/verb on every call, the later Message() was able to remember this for you (by way of GET and NEXT subfunctions) – hence the appeal of recursion. Unless otherwise noted, a missing message causes the functions to copy a dummy message to the output buffer. The sequence number is incremented **after** fetching the message, but not tested for validity until a subsequent call to NEXT.

6.5.1 GetMessage syscall

(GetMessage noun module verb buffer)

Copies the message given by the tuple $\langle module, noun, verb \rangle$ to the output buffer.
Returns: The buffer address

6.5.2 Message syscall

Subfunctions of the Message syscall:

(Message GET module noun verb cond seq buffer)

If *buffer* is non-NULL, copies the message given by the message tuple to the output buffer. Resets any resource-internal recursion (but not the script-controlled ditto).

If *buffer* is NULL, nothing is copied (obviously). Although recursion is followed for purposes of returning a talker value, the stack is left pointing at the tuple given in the parameters – the sequence number is not incremented either. Also affects the LASTMESSAGE subfunction, q.v.

Returns: On success, the talker value associated with the message is returned in both cases. On failure, 0 is returned.

(Message NEXT buffer)

Same as above, except that the message returned is the one following the previously returned one, where "following" means having the immediately following sequence number. If no such message is found, NEXT continues at the message given by the last stack frame (script-controlled recursion being ignored). If the stack is empty, a dummy message is returned.

(Message SIZE module noun verb cond seq)

Returns the size (in bytes) of the message given by the message tuple, including the terminating zero. Does not touch the GET/NEXT stack, although recursion is otherwise handled normally.

(Message REFCOND module noun verb cond seq)

If the indicated message recurses, returns the *condition* it recurses to, -1 otherwise.

(Message REFVERB module noun verb cond seq)

If the indicated message recurses, returns the *verb* it recurses to, -1 otherwise.

(Message REFNOUN module noun verb cond seq)

If the indicated message recurses, returns the *noun* it recurses to, -1 otherwise.

(Message PUSH)

Saves the current resource-internal recursion context on a stack.

(Message POP)

Restores the last resource-internal recursion context.

(Message LASTMESSAGE *tuple)

**tuple* is a pointer to an array. The message tuple of the last proper message is stored in this array. By "proper" I mean that recursion pointers are followed and instances where *buffer* was NULL are ignored.

Table 6.1 SCI and FreeSCI signal bits

Bit #	Name	FreeSCI constant _K_VIEW_SIG_FLAG_ ...	Meaning
0		STOP_UPDATE	A view updating process has ended
1		UPDATED	The view object is being updated
2	noUpd	NO_UPDATE	Don't actually draw the view
3		HIDDEN	The view is hidden from sight. Often, if an actor is supposed to enter and exit a room (such as the guards in the plazas in QfG2), this bit is used. When he's supposed to enter the room, bit 3 in his signal is cleared. When he leaves, bit 3 is set, but his SCI object is not deleted.
4	fixPriOn	FIX_PRI_ON	if this bit is set, the priority of the view never changes (if it isn't, the interpreter recalculates the priority automagically).
5		ALWAYS_UPDATE	
6		FORCE_UPDATE	
7		REMOVE	The view should be removed from the screen (an interpreter bit - its corresponding script bit is bit 3). If bit 3 isn't set as well, the view reappears on the next frame.
8		FROZEN	Deactivates the mover object of the view (it is "frozen" - the view can still turn, however).
9	isExtra	IS_EXTRA	
10		HIT_OBSTACLE	The view hit an obstacle on the last animation cycle
11		DOESNT_TURN	Meaningful for actors only. Means that the actor does not turn, even though he is walking the "wrong way".
12		NO_CYCLER	The view cycler is disabled. This makes the view float instead of walk.
13	ignoreHorizon	IGNORE_HORIZON	
14	ignrAct	IGNORE_ACTOR	Actors can walk in the rectangle occupied by the view. The behaviour of this bit is odd, and best expressed by example. The Guild Master in QfG1 has his bit 14 set. This means that ego (or someone else) can walk all the way to his chair (try sneaking in on him from behind). If we clear this bit, we can't sneak in on him.
15		DISPOSE_ME	The view should be disposed
[a]		FREESCI_PRIVATE	Used as an intermediate result by the interpreter; marks views that are going to have their nsRect/lsRect regions redrawn (for the test in the main draw algorithm's step 17.1., below)
[a]		FRESCI_STOPUPD	View has been 'stopupdated'. This flag is set whenever the view has the STOP_UPDATE bit set, and cleared as soon as it moves again. Stopupdated views are collided against differently than normal views.

[a] This flag is used internally in FreeSCI; it can't be found in the view objects, only in their copies in the dynview widget list.

Chapter 7

FreeSCI

7.1 Basic differences to Sierra's SCI

Sierra's SCI engine, written back in the late 80s, was designed and built to be fast and efficient. Some evil compromises were made (especially in the animation cycle) that sacrificed cleanness for extra cycles. Also, it was designed to use only a very limited amount of memory, which led to more compromises.

The primary design goal of FreeSCI, on the other hand, was Portability. Written in the late 90s, memory constraints were practically nonexistant, since all game data could easily be stored in memory[1]. Thus, resource loading and hunk memory management is of no importance to FreeSCI. The kernel call "Load", which is used to load a resource to hunk space, simply returns the resource identifier of the resource it is supposed to load, as opposed to a pointer to a pointer to hunk memory.

Apart from that, FreeSCI simply abuses the fact that SCI was designed to be used by various different graphics adapters and sound devices. The graphics and sound commands each had to be interpreted by the currently active sound and graphics drivers, and FreeSCI does nothing more than to interpret them in its own way.

Of course, FreeSCI has to accomodate for versions differences between different SCI builds. These are generally minor issues (like the default alignment of text), but they have to be taken care of in one single program, as opposed to several builds as in the case of Sierra's SCI (some SCI games still ship with old versions of the interpreter, because they assume default values that were changed later on).

Finally, there is the built-in debugger. Sierra SCI used a quick and efficient design, while FreeSCI provides a Command-line interface to the debugger, and several additional commands.

7.2 The built-in debugger

7.2.1 Concepts and basic functionality

The built-in debugger takes advantage of a built-in command interpreter (which is not to be confused with the SCI command interpreter). It's appearance is going to vary in between versions (at the time of this writing, it runs on the terminal FreeSCI was started on, in text mode; later versions will likely integrate the debugger to the graphics screen), but all versions of FreeSCI will come with a working debugger[2]. Consult the documentation of your specific release for details on how to invoke it, if it is not activated automatically.

If activated, the debugger is called in between operation fetching and operation execution. It will show the command that is to be executed next, predicting the action done by send, super, and self calls where possible, and displaying any parameters to calling operations. It will also display the current register values and the number of operations that have been executed. It then waits for user input.

In order to simply execute the next operation, execute the "`s`" command. This will do one step of execution. If you want to execute more than one command, invoke "`s [number-of-steps]`". Other ways to step forward are "`snk`" (Step until the Next Kernel function is invoked) and "function/sret/" (Step until the interpreter RETurns from this function).

[1] This is not true for the speech support some of the later SCI1 and SCI32/SCIWin come with, of course. At the time of this writing, SCI1 support is still non-existant, but later versions of FreeSCI will have to allow for dynamical loading of cdaudio resources.

[2] That's what I hope, anyway.

Speaking of functions, the FreeSCI interpreter also keeps a list of the call stack. This is similar to what the Sierra SCI interpreter provides as the "send stack", but it also includes call, calle and callb commands. Please note that callk commands are not included (some kernel functions actually call functions in user space). To display this list, invoke "`bt`". This function will list all calls on the stack, the parameters they carried, from where they were invoked, and the called object[3] and selector (where applicable).

Selectors are not only used for functions, of course, they are also used as variables. To inspect the selectors of the current object, use the "`obj`" operation. Sometimes you might want to inspect how a send operation influenced an object; do so by calling "`accobj`", which will show the selectors of the object indexed by the accumulator register (as used in sends).

For a complete listing of debugging commands, refer to the next chapter.

7.2.2 Debugger commands

The FreeSCI built-in debugger provides the following commands:

7.2.2.1 accobj

The send operation requires a target object, which needs to be stored in the accumulator. This operation makes it possible to check if there is an object at the location indexed by acc, and, if it is, dump the type of object (Class, Object, or Clone), the object's name, and some other interesting stuff (selector names and values, funcselector names and addresses).

7.2.2.2 bpdel (index)

Deletes a breakpoint from the specified index of the list of active breakpoints.

7.2.2.3 bpe (script, index)

Add a breakpoint terminating when the specified exported function of a script is called

7.2.2.4 bplist

Lists all active breakpoints.

7.2.2.5 bpx (method)

Adds a breakpoint to the specified method.

7.2.2.6 bt

Backtrace: Shows the execution stack, bundled with call parameters and selector names where appropriate.

7.2.2.7 classtable

One of the nice things about FreeSCI is that it doesn't hide its class table as Sierra SCI appears to do. With this command, you have the power to unravel the mysteries of classes and superclasses at your fingertips.

7.2.2.8 clear_screen

Clears the screen background from all dynviews, i.e. only picviews, dropped dynviews and the background pic resource are displayed.

[3] This is note quite correct: The object listed is, in fact, the object which is used as the base object for execution. This only makes a difference if the `super` operation is executed, but it may be confusing. Consider it a bug.

7.2.2.9 clonetable

FreeSCI doesn't take Clone()ing lightly. It carefully notes which clone was created and tracks its current position. This function allows you to find them all, and in the darkness bind them.

7.2.2.10 debuglog [mode]

FreeSCI keeps an internal list of flags for specific areas of the game that should be watched more closely. The 'debuglog' command activates or deactivates debug output for each of those areas. Each area is described by a letter; to activate debugging for that area, use "debuglog +x", where x is the area you want to debug. "debuglog -x" deactivates debugging for that area. To activate or deactivate multiple areas, concatenate their single-letter descriptions. Run "debuglog" without parameters to get a listing of all active modes. The modes and describing letters are listed below.

- a The audio subsystem
- b The Bresenham line algorithm functions
- c Character and string handling
- d System graphics display and management
- f Function calls
- F File IO
- g Graphics
- l List and node handling
- m Memory management
- M Menu system
- p The command parser
- s Base setter: Draws the bases of each actor as colored rectangles
- S Said specs
- t Time functions
- u Unimplemented functionality
- * Everything at once. Use with care.

7.2.2.11 die

Exits the interpreter ungracefully.

7.2.2.12 disasm (address) [number]

The debugger is able to disassemble code parts on the fly. Just give it an *address* to dissassemble (and a *number* of commands to dump, if you're feeling bold enough to look at more than one of them simultaneously). Unfortunately, it can only do send prediction and parameter resolution if it is disassembling the PC.

7.2.2.13 dissectscript (script))

Dumps a script resource (with the specified number) and examines it. Lists classes, static objects, relocation tables, and all the other stuff contained in scripts.

7.2.2.14 dm_*

These are dmalloc utility functions. They are described in the dmalloc section below.

7.2.2.15 draw_viewobj (object)

This operation draws the boundaries of the cel described by the indicated SCI object to the screen. The nsRect is drawn in green, the brRect in dark blue, and the position is marked by a small cross in the cel's priority, within a black box.

7.2.2.16 dump (restype, resnr)

Displays a hex dump of the specified resource.

7.2.2.17 dumpnodes (index)

Lists up to *index* nodes of the parse tree.

7.2.2.18 dumpwords

Lists all parser words

7.2.2.19 gfx_current_port

Prints the port ID of the current port.

7.2.2.20 gfx_debuglog [mode]

Toggles debug flags for the graphics driver. Using "+x", the flag 'x' can be enabled, "-x" disables it. Multiple flags can be set at once, e.g. "+abc" or "-abc". With no parameters, all flags currently enabled are displayed. Note that, depending on the graphics driver in use, some flags might not be used. The list of supported flags follows.

- b Basic driver features
- p Pointing device management
- u Screen updates
- x Pixmap operations

7.2.2.21 gfx_draw_cel (view) (loop) (cel)

Draws a single cel to the center of the screen (augmented by the cel's delta-x and -y values). Depending on your graphics driver, you may have to refresh the screen for this to become visible.

7.2.2.22 gfx_draw_rect (x) (y) (width) (height) (color)

Draws a single rectangle to the screen. The `color` parameter describes an EGA color (0-15) which will be the rectagle's color'

7.2.2.23 gfx_drawpic (pic) [palette] [flags]

Renders a pic resource. The *palette* value specifies the pic's palette to use; if not specified, 0 will be assumed. *flags* set any of the pic drawing flags used in the operational layer (see Section 7.5.4.1).

7.2.2.24 gfx_fill_screen (color)

Fills the entire screen (visual back and front buffer) with an EGA color.

7.2.2.25 gfx_free_widgets

This will free the main visual widget and all widgets it contains. Since it essentially invalidates the structured representation of the screen content, this will make the interpreter run into segfaults if you resume. It is intended for memory profiling and heap testing.

7.2.2.26 gfx_print_dynviews

Prints the current dynview list. This list is generated by the Animate() kernel call and represents the visual state of all dynamical images on the screen. Documentation regarding the meaning of the widget descriptions can be found in Section 7.5.4.3.

7.2.2.27 gfx_print_port [port]

Dumps the contents of the port specified (or, if omitted, the current port) to the output stream. Documentation regarding the meaning of the widget descriptions can be found in Section 7.5.4.3.

7.2.2.28 gfx_print_visual

Prints the visual widget, and, recursively, its contents; this widget is the root widget, therefore, the structured representation of all graphical information will be print. Documentation regarding the meaning of the widget descriptions can be found in Section 7.5.4.3.

7.2.2.29 gfx_widget [widget]

(This function is only available if the interpreter was compiled with widget debugging enabled)

If the parameter is not specified, this will print a list of all used widget debug slots (each widget goes into exactly one slot); if the parameter is specified, it is used as an index in the widget debug slot list, causing the corresponding widget to be print. Documentation regarding the meaning of the widget descriptions can be found in Section 7.5.4.3.

7.2.2.30 gfx_priority [priority]

If no parameter is supplied, the start and end values of the priority line list will be print. Otherwise, this function prints the first line of the specified priority region.

7.2.2.31 gfx_propagate_rect (x) (y) (width) (height) (buffer)

Propagates a rectangular zone from the back buffer (0) or static buffer (1) to the next higher buffer.

7.2.2.32 gfx_show_map [nr]

Draws one of the screen maps to the visual back buffer and updates the front buffer. The maps are numbered as follows:

0 Visual buffer
1 Priority buffer (z buffer)
2 Control buffer

Buffers 1 and 2 will be rendered in EGA colors, with color values representing the associated priority/control values (this is identical to Sierra SCI behaviour).

7.2.2.33 gfx_update_zone (x) (y) (width) (height)

Propagates a rectangle from the back buffer to the front buffer; the rectangle's origin and dimensions are passed as parameters.

7.2.2.34 gnf

Lists the rules of the GNF grammar used internally in FreeSCI to parse input.

7.2.2.35 go

Deactivates debug mode and runs the game. Debug mode can be re-activated in the usual ways.

7.2.2.36 heapdump (address) (number)

Invoking this function will spit out `number` bytes, starting at `address`.

7.2.2.37 heapdump_all

Prints all heap segments, including information whether they are allocated or not.

7.2.2.38 heapfree

Dumps a list of the free heap space (free, not gratis).

7.2.2.39 heapobj (address)

This is the same as `accobj`, but it can interpret any object on the heap. Note that the "home" address of objects (as used here) are 8 bytes into the object structure (which starts with the magic number 0x34 0x12), and points to the first (zeroeth?) selector.

7.2.2.40 hexgrep (resource, hex 2-tuples+)

Searches for a list of hexadecimal numbers inside a single resource (if specified like "script.042"), or in a set of resources (if specified like "pic").

7.2.2.41 list (string+)

If called without parameters, it lists all things it can list. Among these are:

`vars`	Global interpreter variables
`cmds`	All available commands
`restypes`	All resource types
`selectors`	All selectors
`syscalls`	All kernel functions
[resource]	All resources of that type (e.g.: "list view")

7.2.2.42 list_sentence_fragments

Lists all parser rules in their normal almost-CNF representation.

7.2.2.43 listinfo (address)

So FreeSCI doesn't have an interactive list debugger as in Sierra SCI. But it has something better[4]: A list dumper, which lists all list elements, keys, and heap positions.

7.2.2.44 man (command)

Shows a short descriptive message to the command.

7.2.2.45 meminfo

Prints information about heap and hunk memory allocation.

7.2.2.46 obj

This is, in essence, the same function as `accobj`, but it checks the current base object as opposed to the object indexed by the accumulator.

7.2.2.47 objs

Lists all objects, classes, and clones that are currently on the stack. They are identified by their properties, and prefixed with an asterisk ('*') if they are clones, or a percent sign ('%') if they are classes.

7.2.2.48 parse (string)

Attempts to parse a single string, and displays the word groups, word classes and the resulting parse tree, if successful.

7.2.2.49 print (variable)

Prints the contents of one global interpreter variable.

[4] Well, this is debatable.

7.2.2.50 quit

Exits the interpreter gracefully, by shutting down all resources manually.

7.2.2.51 redraw_screen

This function retrieves the background picture, puts it on the foreground, and redraws everything. It's not inherently useful, though.

7.2.2.52 registers

This function will show the current values of the program counter, the accumulator, the frame pointer, the stack pointer, the prev register, and the &rest modifier. It will also print the addresses of the current base object, of the global variables, and of the stack.

7.2.2.53 resource_id (number)

FreeSCI packs resource type and number into the usual resource id combination. Use this little helper function to unpack it.

7.2.2.54 restart [string]]

Forces a restart of the current game. The string parameter is meaningless now.

7.2.2.55 restore_game (name)

Tries to restore a game state from the specified directory. See Section 7.4 for details about this.

7.2.2.56 s [number]

This function will execute *number* steps, or one if number was not specified.

7.2.2.57 save_game (name)

Saves the current game state to a directory with the specified *name*. The directory is created automatically; everything inside is deleted, and the game data is stored. See Section 7.4 for details about this.

7.2.2.58 sci_version

Prints the SCI interpreter version currently being emulated

7.2.2.59 scripttable

Lists all scripts that have been loaded, their positions in memory, and the position of their local variables and exports.

7.2.2.60 se

Steps forward until an SCI keyboard event is received.

7.2.2.61 set (variable, int)

Sets the specified variable to a new value.

7.2.2.62 set_acc (number)

Frobbing the accumulator is not recommended, but it may be fun at times. Use this command to set your favourite register to an arbitrary value and watch things blow up.

7.2.2.63 set_parse_nodes

Sets the nodes of the parse tree, and shows the result in list representation. Useful to display information gathered from a certain hacked version of Sierra's SCI interpreter in a more readable fashion.

7.2.2.64 set_vismap (mapnr)

Sets the visual display map. Mapnr can be any of the following:

0 Visual map
1 Priority map
2 Control map
4 Auxiliary map [5]

This function is a no-op since FreeSCI 0.3.1.

7.2.2.65 simkey (keynr)

Simulates a keypress of a key with the specified key number. Modifiers are not applied.

7.2.2.66 size (restype, resnr)

Displays the total byte size of one single resource.

7.2.2.67 snd ...

This executes a sound command. Due to the nature of pipelining between the sound server and the interpreter, it is possible that the result messages of those operations will not be print immediately, so you may have to issue a second command in order for the results of the first command to be displayed.

Also, please note that after entering the debug console, the sound server is, by default, suspended, so you will have to issue an explicit **`snd resume`** to do anything useful.

7.2.2.68 snd stop

Suspends the sound server. This is the opposite of 'snd resume'.

7.2.2.69 snd resume

Resumes the sound server after it has been suspended.

7.2.2.70 snd play (song)

Instructs the sound server to play the indicated song with a handle of 42.

7.2.2.71 snd mute_channel (channel)

Mutes the indicated MIDI channel; events sent to this channel will be discarded before they reach the sound hardware.

7.2.2.72 snd unmute_channel (channel)

Undoes a previous 'mute_channel' command, or part of a previous 'snd mute'

7.2.2.73 snd mute

Mutes all channels (as per 'snd mute_channel')

7.2.2.74 snd unmute

Unmutes all channels (as per 'snd unmute_channel')

7.2.2.75 snd solo (channel)

Mutes all but one channel

7.2.2.76 snd printchannels

Lists all channels, and the instruments currently playing on them

7.2.2.77 snd printmaps

Prints the instrument names and all General MIDI mappings for the song currently playing. This operation will only work correctly if MT-32 to General MIDI translation is being performed.

7.2.2.78 snd songid

Retreives the numerical ID of the song currently playing from the sound server. Songs started with 'snd play' have a song ID of 42.

7.2.2.79 sndmap ...

Executes MT32 to GM sound mapping commands.

7.2.2.80 sndmap mute (instr)

Mutes the specified instrument

7.2.2.81 sndmap percussion (instr) (gm-percussion)

Maps the specified instrument to a GM percussion instrument

7.2.2.82 sndmap instrument (instr) (gm-instrument)

Maps the instrument to a normal GM instrument

7.2.2.83 sndmap shift (instr) (shift-value)

Sets the shift value for the instrument

7.2.2.84 sndmap finetune (instr) (val)

Fine-tunes the instrument, as via the MIDI command

7.2.2.85 sndmap bender (instr) (bender)

Chooses a bender range for the instrument

7.2.2.86 sndmap volume (instr) (vol)

Sets a relative instrument volume, ranging from 0 to 128.

7.2.2.87 snk [number]

Another step command: Step until the interpreter hits a callk command. If you're hunting for a very specific kernel call, just add its number as a parameter. Syscall hunting has never been so easy.

7.2.2.88 so

"Steps over" one instruction, i.e. continues executing until that instruction has been completed (useful for send, call, and related functions)

7.2.2.89 sret

Step until RETurning. If you're bored of the function you're debugging, just invoke this command. It will step forward until the current function returns.

7.2.2.90 stack (number)

Can't remember what you pushed on that stack, and in which order? This command will display as many stack elements as you want, starting at the TOS.

7.2.2.91 version

Displays the interpreter and SCI game versions

7.2.2.92 viewinfo (number)

Examines the specified view resource and displays the number of loops it has, the number of cels for each loop, and the size for each cel.

7.2.2.93 vmvarlist

Lists the heap positions of the current global, local, parameter, and temporary variables.

7.2.2.94 vmvars (type) (index) [value])

Reads or sets a global, local, temporary, or parameter value. Type must be any of 'g', 'l', 'p', 't', to select global, local, parameter or temporary variables (respectively), while index represents the variable index. If value is not provided, that variable will be displayed; otherwise, it will be set to value.

7.2.3 Console interaction with dmalloc

The FreeSCI console proivdes an interface to the dmalloc memory debugger/profiler, if the interpreter was compiled with dmalloc support enabled. The following commands are provided:

7.2.3.1 dm_log_heap

Prints the current heap state into the dmalloc log file

7.2.3.2 dm_stats

Prints memory usage statistics to the output file

7.2.3.3 dm_log_unfreed

Lists unfreed pointers in the dmalloc output file

7.2.3.4 dm_verify (pointer)

Verifies a pointer and prints the result to the dmalloc output file. Specifying 0 instead of a pointer will verify all pointers currently known to dmalloc.

7.2.3.5 dm_debug (mode)

Sets the dmalloc debug flags (please refer to the dmalloc documentation for a description)

7.2.3.6 dm_mark

Gets a mark describing the current heap situation (see also 'dm_chmark')

7.2.3.7 dm_chmark (mark)

Compares a mark retreived by 'dm_mark' with the current heap situation, and prints the results to the dmalloc output file.

7.2.3.8 dm_print (output)

Prints arbitrary output to the dmalloc output file

7.3 Header files

This section explains what some of the header files are good for.

7.3.1 Core headers

The following headers provide what should be considered core functionality:

7.3.1.1 scitypes.h

This file, included from `resource.h`, provides some of the basic types used in FreeSCI, including some of the types used for specific functions, but also the gu?int(8|16|32) types, which provide (unsigned) types for 8, 16, and 32 bits.

7.3.1.2 resource.h

The main OS abstraction header file; includes `scitypes.h` and provides functions for the following: Queues, memory checks, time inspection, directory traversion, case-insensitive file opening, the 'sciprintf()' function, which is the primary output function in FreeSCI, functions to retreive the user's home directory and the cwd, to create a complete path in the file system, yield to the scheduler (where possible) or trigger a breakpoint.

7.3.1.3 sci_conf.h

In here, the configuration options (as parsed from the ~/.freesci/config file) are listed in a structure; includes function definitions for handling configuration.

7.3.1.4 versions.h

Lists certain SCI versions and functions/macros to examine these versions. Some kernel functions have bugs or changed their behaviour in some versions of SCI; these version numbers should be listed in this file.

7.3.1.5 sciresource.h

Provides definitions, strings, and functions for SCI resource management, including the resource manager function prototypes.

7.3.1.6 sci_memory.h

Prototypes for the sci_alloc(), sci_free() etc. functions for memory management, plus the debug switches available for them.

7.3.1.7 console.h

Prototypes for the SCI console, including functions to hook up SCI console functions and variables.

7.3.1.8 sbtree.h

This header file is only used by the gfx subsystem right now. It provides statically generated binary trees.

7.3.2 VM headers

The most central VM header file is `engine.h`, which contains the state_t structure and several global definitions related to savegame and general path management. This file includes a number of other headers, including the following core VM ones.

7.3.2.1 script.h

Provides definitions for opcodes and script segment types

7.3.2.2 vm.h

Definitions for handling objects on the heap, script and class objects, the selector map, execution stack and breakpoint typedefs, a few global variables for debugging the VM, functions for initializing and running it, for looking up selectors in an object, to save and load the game state and pretty much everything else that involves running SCI scripts.

7.3.2.3 heap.h

Prototypes and definitions for FreeSCI's SCI heap implementation.

7.3.2.4 vocabulary.h

This header file provides definitions and declares functions for decoding vocab resources, from parser rules to VM opcode names to selector names. It also lists explains the functions used for parsing.

7.3.2.5 kdebug.h

Provides the SCIkdebug() and SCIkwarn() functions (and their arguments) for selectively debugging kernel functions.

7.3.2.6 kernel.h

Provides GET_HEAP(), PUT_HEAP(), GET_SELECTOR() etc., also predicates to determine whether heap objects are lists and objects, and a generic text resource lookup function that distinguishes between heap text data and text resources. Also includes priority band information, view signals, and other definitions for kernel functions, plus a listing of all kernel functionality.

7.3.2.7 menubar.h

In here, functions for handling menu bar objects are described, as are a number of constants and values that can be used to customize menu bar displaying. The menubar functions call some gfx functions, but are themselves called from the kernel's menubar handling functions.

7.3.2.8 sci_graphics.h

Provides the SELECTOR_STATE and MAX_TEXT_WIDTH definitions for a number of graphical kernel functions.

7.3.3 Graphics subsystem headers

The gfx subsystem's functionality is described in Section 7.5. Most of the header files it uses are prefixed with "gfx_".

7.3.3.1 gfx_system.h

Provides debug functionality, the core data types (points, rectangles, pixmaps, etc), rectangle and point operations (inlined) and enums and definitions for more complex functions.

7.3.3.2 uinput.h

Describes input events (type, modifiers, etc).

7.3.3.3 gfx_driver.h

Documents the gfx_driver_t structure, and the functions and capability flags it can/must provide.

7.3.3.4 gfx_options.h

This file covers configuration options that can be provided to the gfx subsystem's operational layer. It defines a structure that is also used by `sci_conf.h`.

7.3.3.5 gfx_widgets.h

Describes graphical widgets and the functionality they provide, including constructors for each widget.

7.3.3.6 gfx_state_internal.h

This file covers the "hidden" (non-public) part of graphical widgets and includes many gory details regarding their implementation.

7.3.3.7 sci_widgets.h

Provides more complex widgets that are specific to the needs of SCI.

7.3.3.8 gfx_tools.h

Provides utility functions, primarily for gfx driver writers, but also some functions used in the operational layer.

7.3.3.9 gfx_resmgr.h

Describes the gfx subsystem's resource manager's functions, as used by the operational layer, and prototypes for functions implemented by the interpreter specific part.

7.3.3.10 gfx_resource.h

Functions for operating on gfx resources in general, and also functions for loading/drawing particular resources.

7.3.3.11 gfx_operations.h

Describes the operational layer of the gfx subsystem. Provides an extensive set of 2D graphics functionality.

7.4 Savegames

FreeSCI attempts to store savegames portably; for this reason, most of the game data is saved as plain text, while the graphics are written to png[6] files.

[6] Portable Network Graphics. A very portable graphics format with lossless compression, a free reference implementation, and dozens of useful features.

7.4.1 Savegame directory policy

The general FreeSCI directory policy is simple: If there is a `$HOME`, use `~/.freesci/[game name]/` as your playground, if there is no home, use the current working directory. Savegames are true to that policy. Each save game has a directory associated with itself, and this directory is relative to the directory mentioned above. For example, if you execute "`save_game frobnitz`" in SQ3 on your *BSD box while your `$HOME` is set to `/home/rogerw`, the save game files would be written to `/home/rogerw/.freesci/SQ3/frobnitz/`.

7.4.2 Files

7.4.2.1 state

This is the main save file. It contains huge amounts of text data, which are an almost-complete replication of the game internal state_t structure. The code used to read and write this file is automatically generated by a script called cfsml.pl, and it is believed to be rather flexible; i.e. you should be able to insert blank lines, comment lines, (Using the hash ('#') sign), move assignements around, and change values. The identifiers used in this file are identical to the identifiers used in the c code.

7.4.2.2 heap

This is a binary copy of the heap data. Heap data is internally structured to be identical to SCI heap data (little endian, 16 bit), so it is portable to all platforms.

7.4.2.3 hunk*

These files contain raw hunk data. SCI code may allocate raw hunk data, but it can't do anything with it (except unallocate it again). It is unlikely that you are going to encounter a hunk file in normal SCI code. This may change for later SCI versions.

7.4.2.4 song.*

Songs stored by the sound subsystem.

7.4.2.5 sound

Contains the state of the sound subsystem. The syntax is identical to the used in the "state" file.

7.4.2.6 *.id

Savegame name file for one SCI game. The file names are chosen by taking the game's "unique" identifier and appending a suffix of ".id". This file contains the savegame name in plain text.

7.4.3 Obsolete files

The following files were generated by earlier versions of FreeSCI, but are no longer used:

7.4.3.1 *map.png

The four maps of the main picture are stored in four separate png files:
visual_map.png
priority_map.png
control_map.png
auxiliary_map.png
The meanings of those files should be rather obvious.
visual_map.png contains regular palette or color information, so it is, in fact, a screenshot of the game (the mouse pointer is not shown, since it is not stored in the display maps). The other three png files each contain a greyscale gradient palette.

7.4.3.2 buffer*

These are png files containing the various graphical buffers used in the game. buffer_x.1 is the visual buffer, buffer_x.2 is the priority buffer, and buffer_x.4 is the control buffer. Any combination of these three buffers is possible.

Control and priority buffers contain a grayscale gradient palette.

7.4.4 Caveats

FreeSCI's file saving and restoration functionality isn't perfect. Please be aware of the following flaws and limitations before you dig out your flame thrower:

7.4.4.1 File handles

Open file handles are NOT stored or loaded. If you try to save the game with the built-in debugger while file handles are still open, you will be warned about this and saving will abort, unless you preceed your save directory name with an underscore ('_').

7.4.4.2 Kernel functions

SCI kernel functions are able to call the virtual machine. In practice, this means that you may have two or more vm function calls on your system stack; it is not easily possible to store the game state in this case. FreeSCI does not allow it, and, as far as I know, no Sierra SCI code ever tries to do that.

To determine whether or not this applies to you, run "bt" in the debugger; the "base" number in the first line must be zero, or you won't be able to save the game (restoring should work, though).

7.5 The graphics subsystem

Christoph Reichenbach, April 2nd, 2000

Up until version 0.3.0, FreeSCI used a graphics subsystem which used per-pixel operations on three 320x200 8 bit buffers. This concept, while being simple to implement for driver writers, proved to have several disadvantages:

Non-native memory layout: Using a fixed 8bpp visual buffer meant that, for each update, all graphics would ha
No use of accellerated drawing functions: Many of the targetted graphics drivers supported hardware-accelerat
Scalability moved to the drivers: Each driver would have to take care of magnifying the resulting picture by itse
Manual graphics buffer access: This was in fact used in many places, making it hard to keep track of modificati

Combined with some cases of code rot, these problems suggested a re-write of the complete graphics subsystem, and a more modular re-design in preparation for supporting later revisions of SCI (and, possibly, related engines such as AGI).

This documentation section will describe the architecture and functionality of the new graphics subsystem, which has been in operation since FreeSCI 0.3.1. I will start by giving a general overview of the various components involved and how they interact, and then give a more detailed description of each of those components in sequence.

7.5.1 Architecture

In extension of the architecture used up until FreeSCI 0.3.0, the new graphics subsystem now uses a total of six buffers:

Map Name	# of buffers	scaled	bpp
visual	3	yes	determined by driver
priority	2	yes	8
control	1	no	8

Of these, the visual and priority buffers have to be provided by the graphics driver, since they are relevant for display and may actually be present physically (since the priority map is nothing other than a Z buffer). The control map, a special buffer used by the interpreter to check whether moving objects hit obstacles on the screen or touch zones with special meanings, is only relevant for the interpreter and therefore handled one level above the graphics driver.

I will refer to the level above as the "operational layer". This layer handles all of the primitive graphical operations. It performs clipping, keeps track of modified regions, and emulates functions required but not supported natively by the graphics driver.

The operational layer is also responsible for the four pixmap operations, which draw background pictures, images, text, or mouse pointers. These pointers are only referred to by their respective ID numbers; they are retreived from the graphical resource manager. This graphical resource manager (GRM) is another separate subsystem- it retreives graphical resources in one of a set of standard formats, and translates them to the graphics driver's native format in one of several possible ways. It also receives hints from the operational layer to improve its caching strategy.

Finally, above the operational layer, another layer is situated: This widget layer provides abstract descriptions of things on the screen as objects, so-called widgets. It provides the primary interface for the interpreter to interact with.

7.5.2 Standard data types

There are a number of standard data types defined in `src/include/gfx_system.h` which are used all over the place in the graphics subsystem; therefore, they warrant some special attention in order to understand how it works.

7.5.2.1 point_t

This data type is nothing more than a tuple (`x`,`y`). It describes a coordinate on the screen; a one-line way to generate a point_t is to use the function `gfx_point(x,y)`.

7.5.2.2 rect_t

This type describes a rectangular area on the screen, as a four-tuple (`x`,`y`,`xlen`, `ylen`), where the point (`x`, `y`) describes the upper left point of the rectangle, whereas `xlen` and `ylen` are the number of pixels the rectangle extends to the right on the x and downwards on the y axis, respectively. A rect_t can be generated in-line by the function `gfx_rect(x,y,xl,yl)`.

A number of functions are available to operate on rect_ts. These functions are 'pure' in the functional sense, meaning that they do not modify the original rectangle, but, rather, return a new one (of course, an optimizing compiler will make this a moot point from a performance perspective).

gfx_rect_equals(rect_a, rect_b) This function is a predicate that returns non-zero iff `rect_a` describes the same rectangle as `rect_b`.

gfx_rect_translate(rect, point) Returns a rectangle which equals `rect` translated (moved) by the (`x`, `y`) tuple described by the `point` parameter (i.e. `point` is interpreted as a relative coordinate).

gfx_rect_subset(rect_a, rect_b) A predicate to determine whether all pixels contained in the area described by `rect_a` are also contained in the area described by `rect_b`. Reflexive and transitive.

gfx_rects_overlap(rect_a, rect_b) A predicate to test whether there exists a pixel in the area described by `rect_a` which is contained in the area described by `rect_b`. Reflexive and symmetric.

gfx_rects_merge(rect_a, rect_b) Returns the smallest rectangle containing both `rect_a` and `rect_b`.

7.5.2.3 gfx_pixmap_color_t

This structure describes a single color in a pixmap. It consists of 8 bit `r`, `g`, `b` values to describe a color; when used in a pixmap, it is part of a palette of gfx_pixmap_color_ts where the entry at index `i` describes the color of the respective color index `i` inside the pixmap.

In palette mode, the `global_index` entry is used to store the color index entry of the global palette that correlates with the pixmap index (or `GFX_COLOR_INDEX_UNMAPPED` if this value has not been determined yet).

7.5.2.4 gfx_color_t

gfx_color_t structures contain color information for all three color maps. They consist of a gfx_pixmap_color_t structure, `visual`, which describes the effects of the color on the visual map, an `alpha` entry to describe the color's transparency (0 means 'opaque', 255 means 'totally transparent', although graphics drivers may choose to slighly alter those meanings for performance considerations), `priority` and `control` values for the respective maps, and a `mask` to determine the maps affected.

This mask is a bitwise-OR of the constants `GFX_MASK_VISUAL` (meaning "draw to the visual map"), `GFX_MASK_PRIORITY` ("draw to the priority map") and `GFX_MASK_CONTROL` (guess).

7.5.2.5 gfx_mode_t

The FreeSCI graphics subsystem only supports a small subset of all possible graphics modes; specifically, it only supports modes where the integer value of each pixel can be stored in 8, 16, 24, or 32 bits. Color index mode is supported, but non-indexed mode has additional requirements: Each color aspect of red, green, and blue must be represented by a consecutive sub-vector $<v_c, v_{c+1}, \ldots, v_{c+n-1}>$ of the total color vector $<v_0, v_1, \ldots, v_{b-1}>$, where n and c are non-negative integers, and $c + n \leq b$ holds. With v_b being the most significant bit of the total bit vector, we also require that for each `m` where $0 <$ m $<$ n the bit v_{c+m} should, if set, increase brightness about twice as much as setting v_{c+m-1} would. This allows us to represent each color aspect by means of an AND bitmask and an integer shift value.

This, along with a global palette and the scaling factors, is the core of the gfx_mode_t data. It also contains a shift values and an AND bitmask for alpha values; if these values are set to non-zero by the graphics driver, alpha channel information will be written to the same block of data the color values are written to when pixmaps are calculated. If they are not set, a separate 8bpp alpha data block will be added to the pixmaps.

7.5.2.6 gfx_pixmap_t

The gfx_pixmap_t structure is another fundamental element of the graphics subsystem. It describes a single pixmap, such as a background picture, a cel, a mouse pointer, or a single line of text. It contains up to two references to the graphical data it describes: One unscaled block of color-indexed data (`index_data`, and another block scaled and in the graphics driver's native format (`data`).

Each pixmap contains a local palette of `colors_nr` gfx_pixmap_color_t entries, called `colors`. This palette is allocated dynamically and may be NULL if no `index_data` block is present.

Also, a tuple (`xoffset`, `yoffset`) describes the pixmap's 'hot spot'. This is a relative offset into the unscaled data; it is used to describe the point which drawing operations will refer to. This means that pixmap draw operations on this pixmap will cause it to be drawn `xoffset` pixels (unscaled) to the left of the coordinate specified.

Next comes the unscaled pixmap data, called `index_data`, which occupies a size of `index_xl` * `index_yl` bytes. Each byte is either a reference into the palette, or `GFX_COLOR_INDEX_TRANSPARENT` (0xff), which means that it describes a transparent pixel, unless 256 colors are indeed present in the palette[7]

The pointer `data`, unless NULL, points to a block of data allocated to contain the translated graphical data in the graphics driver's native format. The number of bytes per pixel equals the `bytespp` property of the gfx_mode_t structure it was allocated for, whereas its horizontal and vertical extensions are stored in the `xl` and `yl` properties. Unless the graphics mode indicated that it supports an alpha channel itself, a separate `alpha_map` is also provided, at 8bpp.

Each pixmap also comes with a `pixmap_internal` block, which may be used by graphics drivers to store internal information (like pixmap repository handles).

Finally, each pixmap comes with a set of `flags` with the following meanings:

- `GFX_PIXMAP_FLAG_SCALED_INDEX`: The pixmaps index data is already scaled; any algorithm for calculating `data` (and, possibly, `alpha_map`) therefore must not scale it again.
- `GFX_PIXMAP_FLAG_EXTERNAL_PALETTE`: The palette supplied with the pixmap is stored externally, meaning that it must not be freed when the pixmap itself is freed

[7] This may cause a problem for SCI1 support, which explicitly allows for 256 separate colors to be used alongside with transparency. Possible solutions include a separate transparency bitmap or increasing the number of bits per `index_data` entry to 16bpp.

- GFX_PIXMAP_FLAG_INSTALLED: The pixmap has been installed in the pixmap repository (used by the operational layer, although graphics drivers may choose to verify this if they don't trust that layer
- GFX_PIXMAP_FLAG_PALETTE_ALLOCATED: (only relevant for color index mode) The pixmap's palette colors have been allocated in the internal palette listing and have been set appropriately in the palette
- GFX_PIXMAP_FLAG_PALETTE_SET: (only relevant in color index mode) The pixmap's palette colors have been propagated to the graphics driver
- GFX_PIXMAP_FLAG_DONT_UNALLOCATE_PALETTE: (only relevant in color index mode) Instructs the pixmap freeing operations not to free the palette colors allocated by the pixmap. This is used in cases where the palette is stored externally.

src/include/gfx_tools.h defines many functions for creating pixmaps, allocating index data blocks, copying pixmap regions etc.

7.5.2.7 gfx_bitmap_font_t

These structures provide a bitmap lookup table intended for up to 256 entries. In practice, they are used to store font data. There is little surprising about this structure, with the possible exception of the difference between the height and line_height variables: height describes the actual character size, whereas line_height only describes how many pixels the text rendering functions should leave in between text lines.

7.5.3 Graphics drivers

Every FreeSCI graphics driver provides an individual implementation for one specific target platform, such as the X Window System. In order to work correctly, it needs to implement the interface outlined in src/include/gfx_driver.h and list itself in src/include/gfx_drivers.h. Drivers have some freedom in determining which features they want to provide and which they want to have emulated. These features are determined by flags contained in its variable capabilities.

Graphics drivers must provide at least five buffers: Both priority buffers, and the three visual buffers. They are grouped in three sets labelled the Front Buffer (only one visual buffer), the Back Buffer, and the Static Buffer (both containing both a priority and a visual buffer). Most graphical operations operate on the back buffer, with their results being propagated to the front buffer by means of explicit buffer operations[8].

Driver implementations with limited or no hardware accelleration support, such as those operating on plain frame buffers, may use some shared functionality exported for their benefit. Those functions are listed in the appropriate function definitions below.

Unless specified differently, each function must return GFX_OK on success, GFX_ERROR on failure, and GFX_FATAL if and only if a fatal and unrecoverable error occured, such as the target display being closed by external means.

Functions that receive color parameters must respect those parameters' mask values for GFX_MAP_MASK.

7.5.3.1 I/O and debug functionality

For basic input and output, the GFXDEBUG(), GFXWARN() and GFXERROR() macros defined in src/include/gfx_system.h can be used. Also, there is another variable, debug_flags defined for drivers; while it cannot be changed during runtime (yet), it may be used in combination with the various GFX_DEBUG_ constants to selectively enable and disable debugging for certain parts of the driver during development.

For further debugging, the FreeSCI functions sciprintf() (a printf clone), MEMTEST() (tries to detect heap corruption), and BREAKPOINT() (Sets a debugger breakpoint on Alpha, ia32 and SPARC) may be used.

[8] These operations operate on partial buffer contents and expect the back buffer's contents to be unmodified after the transfer. This is unlike the OpenGL back buffer concept.

7.5.3.2 Initialization and shutdown functionality

None of the functions defined in here are optional. They are called during startup or shutdown and need not be considered performance critical.

set_parameter(attribute, value) This function is completely driver specific. Drivers may use it to allow external configuration of options not covered by the standard FreeSCI set of configuration options. It must be implemented to operate correctly both if `init()` has already been called and if it hasn't, although it may choose to ignore any options set afterwards.

Documentation of this function's options is up to the graphics driver's maintainer.

init_specific(xscale, yscale, bytespp) Initializes a graphics driver to a pre-determined mode, where `xscale` and `yscale` are the requested horizontal and vertical scaling factors (integers > 0), and `bytespp` is the number of bytes per pixel on the target display.

The function may set a higher resolution, provided that no matching resolution is available. The `mode` structure (stored locally to the driver structure) must be set by this function if it succeeds; for this, the function `gfx_new_mode()`, defined in `src/include/gfx_tools.h`, may be used.

`GFX_OK` must be returned iff the initialization succeeded; otherwise, `GFX_ERROR` must be reported, unless the graphics target is not (or no longer) able to provide any of the supported modes (e.g. if a required external module was not found, or if the driver detected during run-time that the target does not support any appropriate graphics mode).

init() This operation initializes the driver's default graphics mode. Determining this mode is up to the graphics driver; if its target platform has no means for determining an appropriate mode, it may choose to invoke init_specific() repeatedly with educated guesses. It must return one of `GFX_OK` or `GFX_FATAL`.

See Section 7.5.3.2 for details.

exit() Deinitializes the graphics driver, frees all resources allocated by it and just generally performs clean-up. This function must succeed (so it does not have a return value). It may use `gfx_free_mode()` (from `src/include/gfx_tools.h`) to free the data allocated in the gfx_mode_t structure.

7.5.3.3 Primitive drawing operations

"Primitive drawing operations" here are operations that draw primitives. FreeSCI uses only two graphics primitives: Lines and solid boxes, both of which are commonly provided by graphics libraries. Both operations draw to the back buffer; they also must respect the priority aspect of the primary color used on them.

draw_line(line, color, line_mode, line_style) Draws a single line. The `line` parameter describes the starting point and a relative coordinates of the line to draw in the specified `color`, whereas `line_mode` specifies the line mode to use. This value may be `GFX_LINE_MODE_FAST`, which means that the line's thickness is roughly about the average of the horizontal and vertical scaling factors. The other two values need not be supported (they should fall back to `GFX_LINE_MODE_FAST` if they're used'):

`GFX_LINE_MODE_FAST`: Line thickness is averate of x and y scale factors

`GFX_LINE_MODE_CORRECT`: Lines are scaled separately for x and y and have correct widths there

`GFX_LINE_MODE_THIN`: Line has a width of 1

The other parameter, `line_style`, may be either of `GFX_LINE_STYLE_NORMAL` or `GFX_LINE_STYLE_STIPPLED`, although the latter is used iff the capability flag `GFX_CAPABILITY_STIPPLED_LINES` is set.

This function must return GFX_OK or GFX_FATAL.

draw_filled_rect(rect, color1, color2, shade_mode)

7.5.3.4 Pixmap operations

7.5.3.5 Buffer operations

7.5.3.6 The mouse pointer

7.5.3.7 Palette

7.5.3.8 Event management

7.5.3.9 Capability flag summary

7.5.4 The graphical resource manager (GRM)

7.5.4.1 The operational layer

7.5.4.2 FreeSCI graphical widgets

7.5.4.3 Printing widgets

By means of each widget's `print` method, its state can be written to the FreeSCI output stream. Output of the *STATE* is as follows:

> *STATE* ::= *VALIDITY* "S"*SERIAL ID* [*BOUNDS*] *FLAGS WIDGET-INFO VALIDITY* ::= "v" /* widget is valid */ | "NoVis" /* Valid, but does not have a visual- internal error, unless it's a visual itself */ | "INVALID" /* Widget was invalidated */ | /* empty: Should never happen */ *SERIAL* ::= `HEXNUMBER` /* The widget's unique serial number */ *ID* ::= /* No ID */ | "#"*HEXNUMBER* /* ID assigned to the widget- typically an SCI heap address */ *BOUNDS* ::= (*X-COORDINATE*,*Y-COORDINATE*)(*WIDTH*,*HEIGHT*) /* Full extension of the graphics described by the widget */

The *FLAGS* are described by a sequence of characters; their meanings are listed below:

- `V` Widget is visible
- `O` Widget is completely opaque, i.e. fully covers all area in its bounds
- `C` Widget is a container
- `D` Widget is "dirty", i.e. will be redrawn at the next update
- `T` Widget has been tagged for clean-up
- `M` The widget's ID is not considered to be unique
- `I` Widget will not be freed if a snapshot is resored

The widget's ID will generally be considered to be unique within the container it is appended to, unless the Multi-ID flag ('M') is set. Functionally, this means that a widget `w` is appended to a list containing one or more widgets with an ID identical to its own, it overwrites the first widget with a matching ID, unless `w` itself has the *M* flag set.

The *WIDGET-DESCRIPTION* part of a widget starts with a string describing the widget's type; this is followed by widget- specific information.

7.5.5 Interpreter interaction

7.6 Kernel hacking

Kernel functions are the bridge between the abstract virtual machine, and the world of real programs. The VM may be able to solve RPN equations in the blink of an eye, but what good is this if it can't read input or produce output?[9]

All of the kernel functions are stored in src/core/kernel.c. Since kernel function mapping is done during runtime by string comparison, each kernel function and its name have to be registered in the array kfunct_mappers[]. Note that each version of the SCI interpreter (at least each pre-1.000.000 version) comes with one unidentified kernel function, which is handled by `k_Unknown`.

[9] It could be used to produce benchmarks.

7.6.1 Kernel basics

Each kernel function is declared like this:

```
void
kFooBar(state_t *s, int funct_nr, int argc, heap_ptr argp);
```

So this is how you should start. The four parameters (think of them as the Four Accessories of a kernel function) mean the following:

`state_t *s`	A pointer to the state you are operating on.
`int funct_nr`	The number of this function. Mostly irrelevant.
`int argc`	The number of arguments.
`heap_ptr argp`	Heap pointer to the first argument.

"s" contains a lot of important and interesting data. Have a look at src/include/engine.h for a complete description. What you will probably need mostly will be the heap, (s->heap), a unsigned char pointer, and the accumulator (s->acc), a word (guint16), which is used to return values to the SCI program.

Some kernel functions don't even need to refer to the heap. However, most of them are passed at least one, if not more parameters. This may sound shocking to you, but there is an easy way to work around the neccessity of peeling them off the heap manually: Use the PARAM macros. They are used as follows:

`PARAM(x)`	Returns the value of the parameter *x*. Does not check for validity.
`UPARAM(x)`	Same as `PARAM(x)`, but casts the parameter to be unsigned.
`PARAM_OR_ALT(x, y)`	Checks if `PARAM(x)` is valid and returns it, or returns *y* if `PARAM(x)` is invalid.
`UPARAM_OR_ALT(x, y)`	`PARAM_OR_ALT(x, y)` unsigned.

Several kernel functions assume default values if a specific parameter is not present, to simplify the use of optional parameters. Use the `U?PARAM_OR_ALT(x, y)` macros to detect this case, and you'll rarely have to care about using argc directly.

7.6.2 Hunk and heap

Accessing the heap for both reading and writing is surprisingly important for the kernel, especially when it has to deal with functions that would usually belong into user space, like handling of doubly-linked lists. To ease this, three macros are available:

`GET_HEAP(x)`	reads a signed SCI word (gint16) from heap address x
`UGET_HEAP(x)`	reads an unsigned SCI word (guint16)
`PUT_HEAP(x, foo)`	writes the value foo to the specified heap address

Some kernel functions, especially graphical kernel functions, additionally require the usage of what Sierra referred to as "hunk space". This is dynamically allocated memory; it can even be allocated and unallocated manually from SCI scripts by using the Load() and UnLoad() system calls (this is the sci_memory resource). To allow usage of this kind of memory, three functions have been provided:

`int kalloc(state_t *, space)`	allocate space bytes and return a handle
`byte *kmem(state_t *, handle)`	resolve a handle and return the memory address it points to
`int kfree(state_t *, handle)`	unallocate memory associated with a handle. Returns 0 on success, 1 otherwise

7.6.3 Error handling and debugging

Error handling and debugging probably are the most important aspects of program writing. FreeSCI provides three macros for printing debug output:

`SCIkwarn(text, ...)`	Print a warning message
`SCIkdebug(text, ...)`	Print a debug message
`CHECK_THIS_KERNEL_FUNCTION`	print the function name and parameters

The difference between SCIkwarn and SCIkdebug is that the latter can be easily removed (by commenting out the #define SCI_KERNEL_DEBUG on or about line 39). In practice this means that SCIkwarn

should be used for warning or error messages in cases where it is likely that the vm or the kernel function are doing something wrong; e.g. if the program refers to a non-existant resource file, if a node list command does not come with a pointer to a node list, or if the number of parameters is insufficient. These messages are important and may point to misperceptions of details of the SCI engine. SCIkdebug, on the other hand, is your every-day "flood me with information until I'm blind" debug macro.

Sometimes it may happen that something goes wrong inside the kernel; e.g. a kernel function runs out of memory handles, or an internal variable somehow was set to an invalid value. In this case, `kernel_oops(state_t *, char *)` should be used. It prints an error message and halts the VM, which none of the macros does.

7.6.4 Selectors

Selectors are very important for some of the kernel functions. BaseSetter(), Animate(), Display(), GetEvent() and others take data from or write directly to selectors of a specified object (passed as a parameter or retreived from a node list), or even call object methods from kernel space[10] To prepare the usage of selectors, a variable has do be declared (in src/include/vm.h, selector_map_t). This variable will carry the numeric selector ID during run time. Now, the selector has to be mapped- this is happens once during initialization, to save time. It is performed by script_map_selectors(), which is located at the end of src/core/script.c (just use the "FIND_SELECTOR" macro).

If everything went right, accessing selectors should be really easy now. Just use the GET_SELECTOR(obj, selector) and PUT_SELECTOR(obj, selector, value) macros, where obj is a heap_ptr pointing to the object you want to read from or write to, and selector is the name of the selector to use.

Exemple 7.6.1 An example for PUT_SELECTOR and GET_SELECTOR

```
void
kSwapXY(state_t *s, int funct_nr, int argc, heap_ptr argp)
{
  int posx, posy;
  heap_ptr obj = PARAM(0);

  posx = GET_SELECTOR(obj, x);
  posy = GET_SELECTOR(obj, y); /* x and y are defined in selector_map_t

  PUT_SELECTOR(obj, y, posx);
  PUT_SELECTOR(obj, x, posy);
}
```

Also, it may be neccessary to invoke an actual method. To do this, a varargs macro has been provided: INVOKE_SELECTOR(obj, selector, argc...). In theory, this macro can be used to set and read selectors as well (it would even handle multiple sends correctly), but this is discouraged for the sake of clarity.

INVOKE_SELECTOR works very much like the other macros; it must be called directly from a kernel function (or from any function supplying valid argc, argp and s).

Exemple 7.6.2 An example for INVOKE_SELECTOR

```
INVOKE_SELECTOR(obj, doit, 0); /* Call doit() without any parameters
INVOKE_SELECTOR(s->game_obj, setCursor, 2, 999, 1);
                /* Call game_obj::setCursor(999, 1) */
```

[10] Yes, this is evil. Don't do this at home, kids!