The Level 9 Game Format

Concepts that don’t exist in this document yet, or need more work

- Jumptables
- Lists
    1. How do lists work? Are all list values 8-bit or 16-bit, or can it be a mix?
- Messages
- Strings
- Text compression
- The header
- Line graphics
- Bitmap graphics
- Differences between version 1, 2, 3, 4
- Save files, saving and restoring
- The stack
- The workspace
- Do some variables have special meaning defined by the format?
- What exactly are the parser’s responsibilities? Are words like “the” removed from input? Can the program access all the words that were typed?

Instruction encoding

An instruction starts with a single byte value which we will call the *code byte***.** If bit 7 is clear, this instruction uses one of the 27 basic opcodes, and bits 0-4 hold the number of the opcode. Otherwise, it’s a list handler.

If one of the arguments is a constant, this applies: If bit 6 of the code byte is set, it’s a byte constant, otherwise it’s a word constant. Constants are generally unsigned. Possibly, some functions may treat them as signed. **Are the byte/word constants signed or unsigned? Does it matter? (FS) -- L9dis prints it as an unsigned hex number, so I suppose it’s either unsigned or undetermined. (FR)**

If one of the arguments is an address, this applies: If bit 5 of the code byte is set, the argument is a signed byte, giving a relative address in the range pc - 128 to pc + 127. Otherwise the argument is a little-endian word. **If this address points to somewhere in the jump table area, something happens, not sure what?**

There are 256 global variable slots. A variable argument is always a byte pointing to one of these slots.

Basic Opcodes

These are the basic opcodes:

GOTO *address* 0

Sets pc to address

GOSUB *address* 1

Push pc to stack and set pc to address.

RETURN 2

Pop address from stack and set pc to it.

PRINTNUMBER *variable* 3

Print variable as decimal number.

PRINTMESSAGE *variable* 4

Print the message with the number held in the variable.

When printing the message the first alphabetic character ([a-z,A-Z]) following after a full stop ('.'), a question mark ('?') or an exclamation mark ('!') is automatically converted to uppercase.

PRINTMESSAGE *constant* 5

Print the message with the number given by the constant.

When printing the message the first alphabetic character ([a-z,A-Z]) following after a full stop ('.'), a question mark ('?') or an exclamation mark ('!') is automatically converted to uppercase.

FUNCTIONCALL *functionnumber* [possibly a variable or a string] 6

Functionnumber is a byte value. Executes one of the following functions, depending on functionnumber:

1. **calldriver**
   One source says this quits the game. From reading the Level9 terp source, I think it can call up to 22 different routines. The ID of the routine to call should first be stored in list#9[0], and if this call takes an argument, this is stored in list#9[1]. If the routine returns a value, this value is stored in list#9[1] when it returns.
2. **random** variable

3. **save**
4. **restore**
5. **clearworkspace**
6. **clearstack**
7. **printstring** string
   string is a sequence of characters terminated by a null byte. **Not sure if this uses compression, if it's somehow encrypted or if it's just clear text.**

INPUT *variable1 variable2 variable3 variable4* 7

Read user input and parse it.

In version 1 and version 2 the parsing returns up to three word values in the first three variables specified and the word count in the fourth variable. The input is parsed from left to right and unrecognized words are ignored, but counted in the word count. Word values returned are the values associated with the word found in the dictionary. In *Snowball* (v1), for example, the sentence "pull the lever beside" returns:

```
variable1   64  word value for "pull"
variable2  167  word value for "lever"
variable3    0
variable4    4  word count
```

"the" and "beside" are not in the dictionary and simply ignored. One quirk in the parsing is that matching against the dictionary only matches so long as the inputted word's length. Again examples from *Snowball* (version 1):

```
"a" matches, and returns word value for "again"
"an" matches, and returns word value for "angel"
"x" matches, and returns word value for "xyzzy"
```

This helps the player insofar as the player only needs to type the number of characters that distinguishes the word, but can also give confusing responses as in the example "x".

In version 3 and 4 the variables are ignored and the result of the parsing is returned in list 9.

VARCON *constant variable* 8

Copy the value of constant to variable.

VARVAR *variable1 variable2* 9

Copy the value of variable1 to variable2.

ADD *variable1 variable2* 10

Add variable1 to variable2, storing the result in variable2.

SUB *variable1 variable2* 11

Subtract variable1 from variable2, storing the result in variable2.

ILLEGAL 12

ILLEGAL 13

JUMPTABLE *jumptablestart variable* 14

The jumptable contains a table of little-endian words that forms an address. The jump sets the pc to the address stored at jumptablestart + 2 * variable.
There is no size defined for the table but its maxsize is 256 addresses (the variable is only one byte). The actual size of the table and where the A-code begins again has to be determined indirectly by inspection of **GOTO**, **GOSUB** and other addresses in the A-code when disassembling (this is actually of no concern in already assembled code).

EXIT *variable1 variable2 variable3 variable4* 15

IFEQVT *variable1 variable2 address* 16

If variable1 equals variable2 then goto address.

IFNEVT *variable1 variable2 address* 17

If variable1 does not equals variable2 then goto address.

IFLTVT *variable1 variable2 address* 18

If variable1 is less than variable2 then goto address.

IFGTVT *variable1 variable2 address* 19

If variable1 is greater than variable2 then goto address.

SCREEN *byte* 20

Set screen mode to one of the following depending on byte value:

0. textmode
1. graphmode byte

cleartg *byte* 21

Clear the text or graphics. Clears one of the following depending on byte value:

0. textmode
1. graphmode

PICTURE *variable* 22

GETNEXTOBJECT *variable1 variable2 variable3 variable4 variable5 variable6* 23

IFEQCT *variable constant address* 24

If variable equals constant goto address.

IFNECT *variable constant address* 25

If variable does not equals constant goto address.

IFLTCT *variable constant address* 26

If the variable is less than the constant goto address.

IFGTCT *variable constant address* 27

If the variable is greater than the constant goto address.

PRINTINPUT 28

List handlers

Bits 0-4 of the code byte decide which list is being operated on. Valid values are 1-9 (but not sure all games provide all ten lists). Bits 5-6 determine which list handler is used, as follows:

%00: **listc1v** byte variable
Set list[byte] = variable

%01 **listv1v** variable1 variable2
Set variable2 = list[variable1]

%10 **listv1c** byte variable2
Set variable2 = list[byte]

%11 **listvv** variable1 variable2
Set list[variable1] = variable2

Opcode summary

Basic opcodes

Opcode	Code	Bytes
goto	%0xx00000	2 (a)
gosub	%0xx00001	2 (a)
return	%0xx00010	1
printnumber	%0xx00011	2
printmessage	%0xx00100	2
printmessage	%0xx00101	2 (b)
functioncall	%0xx00110	See separate table
input	%0xx00111	5
varcon	%0xx01000	3 (b)
varvar	%0xx01001	3
add	%0xx01010	3
sub	%0xx01011	3
jumptable	%0xx01110	4
exit	%0xx01111	5
ifeqvt	%0xx10000	4 (a)
ifnevt	%0xx10001	4 (a)
ifltvt	%0xx10010	4 (a)
ifgtvt	%0xx10011	4 (a)
screen	%0xx10100	2 TEXTMODE 3 GRAPHMODE
cleartg	%0xx10101	2
picture	%0xx10110	2
getnextobject	%0xx10111	7
ifeqct	%0xx11000	4 (a) (b)
ifnect	%0xx11001	4 (a) (b)
ifltct	%0xx11010	4 (a) (b)
ifgtct	%0xx11011	4 (a) (b)
printinput	%0xx11100	1

(a) +1 if bit 5 is clear, i.e. it’s using an absolute address
(b) +1 if bit 6 is clear, i.e. it’s using a word-sized constant

Function calls

Opcode	Code	Bytes
calldriver	%0xx00110, 1	2
random	%0xx00110, 2	3
save	%0xx00110, 3	2
restore	%0xx00110, 4	2
clearworkspace	%0xx00110, 5	2
clearstack	%0xx00110, 6	2
printstring	%0xx00110, 7	3+

List handlers

Opcode	Code	Bytes
listc1v	%100xxxxx	3
listv1v	%101xxxxx	3
listv1c	%110xxxxx	3
listvv	%111xxxxx	3

Dictionary

Version 1 and version 2

The dictionary consists of a series of bytes where all characters of each dictionary word are stored. The characters are stored in the default case. and the last character has bit 7 set. After the last character is a byte holding the word value. This is the value returned by the input opcode. Several words can share the same value, e.g. N and NORTH can use 1, S and SOUTH can use 2 etc.

Only characters '!', ''', ',', ' -', '.', '/', '0'-'7', ' ?' and 'A'-'Z' (0x21, 0x27, 0x2c-0x2f, 0x30-0x39, 0x3f and 0x41-0x5a) are allowed in dictionary words.

Version 3 and version 4

Dictionary data is ordered in banks. Each bank can be stored separate from other banks in memory. The number of banks is defined by *dictdatalen* and at *dictdata* each bank is referred to by 4 bytes each. Bank 0 is a special case and has no entry in the dictdata-table. Bank 0 always starts at *defdict* and the word number for the first dictionary word in this bank is always 0. The word numbering is then strictly sequential.

The dictionary words are sorted alphabetically (independent of word casing). Words whose initial character is not a letter must be in the first bank (bank 0). Thereafter each entry in the dictdata-table has the following format:

```
byte 0-1: Little-endian address of the next bank
byte 2-3: Little-endian word number of first entry in this
          bank
```

Words are placed in banks depending on the two first letters:

```
Bank
  0   Words whose initial letter is lower than 'a'
  1   Words beginning with 'aa' to 'ah'
  2   Words beginning with 'ai' to 'ap'
  3   Words beginning with 'aq' to 'ax'
  4   Words beginning with 'ay' to 'az'
  5   Words beginning with 'ba' to 'bh'
  6   Words beginning with 'bi' to 'bp'
...
104   Words beginning with 'zy' to 'zz'
```

This helps the interpreter to quickly locate the word in the dictionary to get its word number.

Example: “`LEVER`” - The character index for `'L'` is 11 and the character index for `'E'` is 4. If we multiply 11 by 4 and shift in the two highest bits from 4 (all numbers are 5-bits long) we get 44. Ergo the word should be in bank 44.

All dictionary words in a bank are stored packed, 5 bits (hereafter called *pentad*), for each character (8 characters in 5 bytes). There is no break between words and a new dictionary word can start in the middle of a byte. The first pentad in each dictionary word should be 28, 29, 30 or 31. Each *pentad* have the following meaning:

```
0-25 The lowercase letters  'a' to 'z'
26   The bits 0-7 of the next two pentads constitutes a
     long code (the actual ASCII). This is for uppercase,
     numbers and other special characters.
     This also have a special case where that if the next
     pentad is 0x10 the rest of dictionary word should be
     printed in uppercase.
27   End of dictionary words in this bank
28   Keep 0 characters from previous dictionary word
29   Keep 1 characters from previous dictionary word
30   Keep 2 characters from previous dictionary word
31   Keep 3 characters from previous dictionary word
```

To illustrate with an example (from *Snowball*, version 3):

```
Hex values from bank 78:
e6 85 4d 05 a9 a3 45 34 39 bc 98 02 b2 79 44 f9 25

Converted to pentads (decimal values):
28                     Keep 0 characters from previous entry
26 2 20                Long code 84 = 'T'
26 1 13                Long code 29 = '-'
9 20 13 2 19 8 14 13   'junction'
28                     Keep 0 from previous
19 0 1 11 4            'table'
30                     Keep 2 from previous = 'ta'
10 4                   'ke'
31 4 18                Keep 3 + 'es' = 'takes'
```

The dictionary words 'T-junction', 'table', 'take' and 'takes' (24 characters) are stored in 135 bits, a little less than 17 bytes, occupying about 70% of its original size.

Header

Version 1

Games in this version don't contain any header information. The information where the different parts are located within the files are coded into the games themselves. For the five known games that exists in this version the start of the A-code (acodeptr) have the footprints and the other parts relative to that as below:

Colossal Cave Adventure

```
Hex footprint: 20 04 00 49 00 06 05 48 01 01 48 02 02 48 ff
03
acodeptr          0x0000
dictdata         -0x0760
absdatablock     -0x03b0
startmd           0x0f80
startabbrev       0x57d7
```

Adventure Quest

```
     Hex footprint: 00 06 00 00 46 00 06 05 48 01 01 48 02 02 48
03
     acodeptr          0x0000
     dictdata         -0x04c8
     absdatablock     -0x0800
     startmd           0x1000
     startabbrev       0x49d1
```

Dungeon Adventure

```
     Hex footprint: 00 06 00 00 44 01 06 05 48 01 01 48 02 02 48
03
     acodeptr          0x0000
     dictdata         -0x0740
     absdatablock     -0x0a20
     startmd           0x16bf
     startabbrev       0x58cc
```

Lords of Time

```
     Hex footprint: 00 06 00 00 65 01 45 a0 08 0f 01 01 48 00 02
48
     acodeptr          0x0000
     dictdata         -0x4a00
     absdatablock     -0x4120
     startmd          -0x3b9d
     startabbrev      -0x0215
```

Snowball

```
     Hex footprint: 00 06 00 00 d4 01 45 a0 48 00 01 48 00 02 48
00
     acodeptr          0x0000
     dictdata         -0x0a10
     absdatablock      0x0300
     startmd           0x1930
     startabbrev       0x5547
```

Version 2

Hex	Contents
0x0000	startmd - Start of message data
0x0002	startabbrev - Start of abbreviation data
0x0004	absdatablock - Is always 0x0020
0x0006	dictdata - Start of dictionary data
0x000a	Unknown - Is always 0x8000

0x0014	Unknown - Always identical to value in 0x0016
0x0016	Unknown - Always identical to value in 0x0014
0x001a	acodeptr - Start of A-code
0x001c	Datafile length

Version 3 and version 4

Hex	Contents
0x0000	Datafile length
0x0002	startmd - Start of message data
0x0004	Length of message data
0x0006	startdict - Start of dictionary
0x0008	Length of dictionary
0x000A	dictdata
0x000C	dictdatalen
0x000E	wordtable - 128 predefined words used for uncompressing messages.
0x0010	unknown
0x0012	absdatablock
0x0014	unknown - L9Dis lists this in its header output but calls it illegal in code.
0x0016	list1
0x0018	list2
0x001A	list3
0x001C	list4
0x001E	list5
0x0020	list6
0x0022	list7
0x0024	list8
0x0026	list9
0x0028	acodeptr - Start of A-code

Checksum

In games of version 2-4 the last byte in the file should be a padding so that when every byte in the file from start of header to the end of the file (datafile length) is added (modulo 256), the resulting byte equals 0.

Messages

Version 1

Messages are made up by two tables; the actual messages starting at *startmd* and the 162 abbreviations starting at *startabbrev*. Both messages and abbreviations start the numbering at 0 and the characters are used as this:

```
0x00       Invalid
0x01       EOS (End-of-String)
0x02       Invalid
0x03-0x5d  0x1d is added and the corresponding ASCII is
           printed, except for '%' (0x25) that is replaced
           by CR (0x0d) and '_' (0x5f) which is replaced by
           SPACE (0x20).
0x5e-0xff  0x5e is subtracted and the corresponding
           abbreviation with the resulting number
           is printed here instead. This works recursively
           for abbreviations.
```

Note that this allows the abbreviations to have parts that are other abbreviations, in other words the abbreviations can be recursive.

```
Message table
M0000 01              ""
M0001 42 01           " "
M0002 01              ""
M0003 0f 42 01        ", "
M0004 11 42 01        ". "
M0005 08 01           "!"
M0006 a2 47 42 01     "[A002]d "

Abbreviation table
A001  44 51 01        "an"
A002  03 6f 01        " [A001]"
```

Version 2

Version 2 either has exactly the same message encoding as version 1 but there is also a different message encoding, maybe a later addition, used on some games on some platforms.

The new encoding have the following differences:

- The length of the message is coded differently. Instead of an EOS indicator the first byte(s) contains the length of the message and then the message starts at the second byte. If the message is longer than 255 characters the second byte is 0 to add 255 to the length and the message starts at the third byte. This can go on as long as it takes to add 255 to the message length and the message starts at the first byte that is not 0. Note that the bytes defining the length are included.

  ```
  06 42 57 4b 48 42          " the "    (len = 6)
  05 00 57 4b 48 42 .. ..    " the ..." (len = 230)
  ```

- Message and abbreviation numbering starts at 1 instead of 0. This has the effect that 0x5d, instead of 0x5e, has to be subtracted for including abbreviations.

- The first abbreviation starts with its length one byte before the address given in the header. This means that the abbreviations start at *startabbrev* - 1

Version 3 and version 4

In version 3 and version 4 messages can have two purposes; the first one is obvious, to hold the text messages for the game. The second purpose is to connect the dictionary words with their synonyms.

The messages start at startmd with message 0 and then each message follows incrementally.

The first byte of the message is either an instruction to skip message numbers in the numbering or the start of the message length information (see next paragraph about the message length). If bit 7 of the first byte is set, this message is skipped and message numbering is increased by the number stored in bits 0-6 plus 1. If bit 7 of the first byte is clear then the length byte(s) begins here.

The length can be defined by one or more bytes. The length is the value of bits 0-5 in this byte. If the value of bits 0-5 of the length byte is 0, the length is 63 + the length in the next byte (bits 0-5), this can be repeated. If bit 6 is set (length byte is 0x40 instead of 0x00) then 1 is subtracted from the total calculated length. If the value of the length byte is greater than 0x40, in other words bit 6 is set but bits 0-5 is clear, this is a message that contains a dictionary word and its synonyms (see below how this is encoded inside the message).

```
80                    MSG-0, skip
02 00                 MSG-1, length = 2
```

```
81                    MSG-2 & MSG-3, skip
02 03                 MSG-4, length = 2
46 05  c6 88 ...           MSG-5, length = 6, synonyms
00 12 8f 8d 87 01     MSG-6, length = 81
40 12 8f 8d 87 01     MSG-7, length = 80
```

After the length is established the actual message follows. The message contains a series consisting of two bytes (big-endian) words or single byte values that point to one of the 128 big-endian words in the *wordtable*. If bit 7 is set then it's a pointer to the *wordtable*, otherwise this byte and the next byte is a big-endian word. The two-byte word obtained either directly or from the *wordtable* is hereafter called *wordref* and contains encoding information.

Bits 0-11 either is a pointer to a dictionary word, an ASCII character or a divider between a word and its synonyms.

- If bits 0-11 of the *wordref* is less than 0x0f80 then the corresponding dictionary word is fetched from the dictionary and printed.
- If bits 0-11 of the *wordref* is greater than 0x0f80 then the character, defined by the ASCII value in bits 0-6, is printed.
- If bit 0-11 of *wordref* is exactly equal to 0x0f80 then the preceding word's synonyms follow after 0x0f80. If this message is printed, only the first part is printed to the screen.

Bits 12-15 of the *wordref* determines how the *wordref* should print.

- If bit 12 is set and bits 0-11 is greater than 0x0f80 then a space should be printed after the *wordref*.
- If bit 13 is set and bits 0-11 is greater than 0x0f80 then a space should be printed before the *wordref*.
- If the value of bits 12-15 is greater than 5 and bits 0-11 is less than 0x0f80 (dictionary word) then print *wordref* in uppercase.

When printing consecutive dictionary words, a space is always printed before the dictionary word. When a dictionary word is following an ASCII character, is the first word or the ASCII character explicitly printed a space after (bit 12 set); no space is printed before the dictionary word.

As in version 1 and version 2 the first alphabetic character ([a-z,A-Z]) following after a full stop ('.'), a question mark ('?') or an exclamation mark ('!') is automatically converted to uppercase.

L9Dis have a scheme to replace international characters. This is according to L9Dis used in "Champions of Raj". This doesn't seem to be implemented in Level9, the terp. Should it be documented? HÅ