The Z-machine, And How To Emulate It

…being a Rewrite of Graham Nelson's Standard 0.2 Specification of the Z-machine

version 0.6e, 28 November 1996
(this PostScript file created on 28 November 1996)

by Marnix Klooster `<marnix@worldonline.nl>`

Notes

– From version 0.6d onwards, this document is written using the document formatting language Lout. This makes it is available in both plain ASCII and PostScript, but not in DVI format.

– Feedback on all aspects of this rewrite is very welcome. These include my (ab)use of the English language, typos, errors, obscurities, structure, and the Questions that appear throughout. Feel free to write me, and get your name mentioned in the introduction!

Major changes from version 0.6 to 0.6a

– Renumbered sections 3-6 to 5, 4, 6, 3.

– Simplified text style handling.

– Simplified the buffering process.

– Completed section 8.

Major changes since version 0.6a

– Introduced modes 'predictable' and 'unpredictable' for the random generator.

– Added an index.

To Do

– Add something on menus.

– Correct fixed/variable-width font selection in V3 (and higher?).

– Check all details against Standard 0.2 [Nelson], the changes described in the Informal Z-machine Newsletter #1, and Stefan Jokisch's list of comments.

– Disallow output characters 1024-65535?

– Add table describing the graphics font; or create a separate document for that; or refer to [Nelson] (but then this document won't be self-contained).

– Complete, correct, and beautify the index.

– Add a diagram describing the Z-machine's structure.

Contents *Page numbers differ between PostScript and text versions!*

0. Introduction 2
1. How To Read This Document 3

2. General Structure And Operation 5
3. Data Structures 11
4. The Video Card 17
5. Characters, Fonts, And Styles 23
6. The I/O Card 27
7. The Structure Of Instructions 30
8. A Catalogue Of Instructions 34
Miscellaneous Tables 52

References 60

Index 62

0. Introduction

The Z-machine is a (basically) elegant imaginary computer that runs text-adventure games. It was conceived and designed by the founders of Infocom, a company that produced a line of high-quality adventure games in the 1980s. By writing these in Z-code, and emulating the Z-machine on many different computers, Infocom succeeded in reaching a wide audience. Their emulators – and a number of free ones – make programs written for the Z-machine very portable. Over the years there has been a large amount of archeology into the world of the Z-machine and its emulators, the main investigators being the InfoTaskForce (ITF), Paul David Doherty, Mark Howell, Matthias Pfaller, and Mike Threepoint. The reason for these investigations is that today, the Z-machine still is a good vehicle for running text adventures – or Interactive Fiction, as the admirers of this form of art like to call it.

Below, I attempt to completely describe the Z-machine, and give hints about its emulation; therefore, this document might be of use to writers of emulators and Z-code alike. I draw heavily on the definitive guide to the Z-machine and its history, Graham Nelson's "Specification of the Z-machine" [Nelson]; in fact, his Specification inspired me to do this rewrite. To steal a quotation from it:

> *"The highest ideal of a translation [...] is achieved when the reader flings it impatiently into the fire, and begins patiently to learn the language for himself."*

Well, here is another translation for you to feed to the flames. I am well aware that I can't hope to match Graham's literary style of writing; but what I lose in poetry, maybe I can make up for in clarity. I wrote this mainly for myself, as a clarifying and concise supplement to his Specification; perhaps others find it useful too. If not, feel free to ignore it. In any case, please let me know what you think.

An alert reader will probably find places where this rewrite contradicts the Specification. In such cases he or she should follow the Specification – and I wouldn't mind being informed of these contradictions.

Many thanks go to Paul David Doherty and Stefan Jokisch for their detailed comments

on earlier versions of this document – which probably never would have existed at all without everything Graham Nelson has done for the interactive fiction community.

There is an electronic mailing list available for discussion of the Z-machine, thanks to the kind people at GMD in Germany. To subscribe, send electronic mail to `majordomo@gmd.de` containing the following text in the *body* of the message:

```
subscribe z-machine YourEMailAddress@Goes.Here
```

Unsubscribing can be done similarly, replacing `subscribe` by `unsubscribe.` For more information on Infocom, Inform (Graham Nelson's compiler targeted at the Z-machine [Inform]), or interactive fiction in general, take a look at the Interactive Fiction Archive [IFA], or read the Usenet newsgroup `rec.arts.int-fiction`.

After section 1, which introduces some terminology, the overall structure of the Z-machine is described in section 2. Section 3 explains the main data structures. The video card is the subject of section 4. Section 5 details the fonts and styles, and section 6 describes how the I/O card uses these. Next, section 7 delves into the structure of instructions, and the last section details the effect of every legal Z-code instruction. It is followed by an appendix containing reference tables, and I close off with a few references to literature and software.

1. How To Read This Document

This document describes the structure and operation of the Z-machine. This is not an easy thing to do, since there are in fact a number of different Z-machines. It all began of course with the Infocom Z-machine, of which six versions were created over the years. After Infocom deceased, there have been a number of attempts and proposals to refine and extend the Infocom Z-machine. The most recent result in this direction is the Standard Z-machine, an attempt to clear up a number of fuzzy issues, at the same time collecting some useful extensions. This document basically describes the Infocom Z-machine, with Standard extensions and changes marked down explicitly as such.

We will refer to the six different versions of the Z-machine as V1 to V6, V*n*+ meaning all versions from *n* to 6. Within each version, there are Z-machines with different capabilities; for instance, some Z-machines can show pictures, while others cannot. The machine language of the Z-machine is called Z-code, which differs a bit between versions. A game file that is to be run on a Z-machine is called a Z-program.

Standard extension: Versions 7 and 8 were invented as non-Infocom extensions by Graham Nelson to accommodate very large Z-programs. These are completely equivalent to V5, except for the version number, the meaning of packed addresses, the maximum program size, and the meaning of the header word at $1A; see 2.3, and the `verify` instruction in section 8.

1.1. Emulators

Since the Z-machine is (still) an imaginary computer, the only way to run Z-programs is by emulating this machine. We distinguish between the Z-machine and its emulators (or interpreters).

It is important to realize that an emulator is only committed to producing the behaviour that is described in here. How this behaviour is achieved, *i.e.*, how the emulator is implemented, is not important for the purposes of this specification. For instance, in the following a distinction is made between a call stack and a routine stack; an emulator might very well use one stack to implement these (as indeed ZIP and Frotz do).

On the other hand some emulators might not be able to completely follow the specification, *e.g.*, for technical reasons. (For example, most emulators probably won't implement the unlimited call and routine stacks discussed below. Or an emulator might just emulate enough of the Z-machine to run one version, or even one specific game file.) In such a case, this deviation should be clearly indicated (*e.g.*, by giving an appropriate message), and if possible a behaviour in the spirit of the Z-machine should be substituted.

1.2. Different behaviours

There are some terms that are used here in a technical sense only: *error*, *unspecified*, and *legal*. If something is called an *error*, then the Z-machine presumably crashes at that point. A friendly emulator would display an error message and halt, but an unfriendly one might crash, or even go on as if nothing has happened. Note that an erroneous construct – be it instruction or data structure – is only an error if it is encountered or used. Sometimes the result of an operation is called *unspecified*, and this is often accompanied by the description of a recommended behaviour. In this case an unfriendly emulator might crash, but a friendly one would behave as recommended, optionally displaying a warning message. Anything that is not an error or unspecified is called *legal*; the effects of all legal operations are completely specified in this document.

The writer of Z-code should avoid anything that is not legal. Of course, any situation set down as illegal in this document, but not occuring in an existing Z-program (*i.e.*, Infocom or Inform game file), could be made legal if the writers of emulators agree on its interpretation.

1.3. Notations

There are a number of terms used to refer to numbers:

bit:	either 0 or 1. In any sequence of bits, the individual bits are numbered from right to left, counting from zero.
bottom (top) bit:	the least (most) significant bit, *i.e.*, the bit with the lowest (highest) number.
byte:	a sequence of 8 bits.
word:	a sequence of 16 bits.
natural number:	0, 1, 2, …
integer:	…, -2, -1, 0, 1, 2, …
unsigned:	a sequence of bits is read as natural number.
signed:	a sequence of bits is read as an integer. Let n be the unsigned natural number represented by the sequence of bits. If the top bit is 0, the sequence is read as n ; if the top bit is 1, the sequence is read as $n - 2^{\text{number of bits}}$.

If x is a sequence of bits, the phrase 'x is byte-valued' is used to mean: x has an (unsigned) value between 0 and 255, or equivalently: all bits from bit 8 upwards are 0.

Any sequence of bytes can be interpreted as a sequence of bits, by taking first the bits of the first byte, then the bits of the second, etc. Conversely, any sequence of a $8 * n$ bits (*e.g.*, a word or a longword) can be split up into n bytes. The top 8 bits form the first byte, the next 8 the second byte, etc.

The percent sign '%' followed by a number of bits stands for that sequence of bits. (Note that [Nelson] and [Inform] use '$$' instead of '%'.) The dollar sign $ followed by a number of hexadecimal digits also stands for a sequence of bits, each digit representing four bits. The notation 'n/m' means bit m at address n.

Another useful piece of notation is the m-n-table, which consists of m bytes representing an unsigned number, followed by that much entries of n bytes long.

Taking a number 'modulo n' (for $n > 0$) means that a multiple of n is added to or subtracted from the number to make it at least 0 and less than n. For example, -535 modulo $10000 equals 65000. The expression floor(n) stands for the greatest integer that is not greater than n. For example, floor$(-3.1415) = -4$.

2. General Structure And Operation

The main components of the Z-machine are the central processing unit (CPU), the memory, the call stack, the random number generator, the I/O card, and the video card; these are connected to the outside world in various ways. From version 3 the Z-machine contains a sound card. From version 4 a timer is added, and different text styles can be used. From version 5 the Z-machine optionally has a save memory, which can be used to revert to a previous state; and non-ASCII fonts may be provided. In version 6 the video card can show pictures, and an optional mouse can be used. This section describes all components; a detailed description of the video and I/O cards is deferred to sections 4, 5, and 6. Finally, the initialisation and operation of the Z-machine are described.

2.1. CPU

The CPU is the spider in the web: it is connected to memory, call stack, random number generator, video card, and the V3+ sound card. The CPU reads instructions from memory, and then decodes and executes them. It is possible to interrupt the CPU by giving the address of a routine (see 3.7) to be executed. In this case control is passed to the interrupt routine, and the result of this routine is returned to the caller. Note that the CPU does not contain a program counter or any registers; these are found at the top of the call stack.

2.2. Call stack

The call stack is a stack of frames, which is unlimited in size. Each frame contains a program counter (PC), up to 15 local variables and a routine stack, and also some administrative information in V3+. On start-up a single frame is put on the call stack. When a routine is called a newly created frame is pushed on top, passing control to the called routine. On a return instruction, the top frame is removed and the previous frame comes on top again. Only the top frame can be accessed and modified; it gives the current state of the Z-machine. A frame consists of the following elements:

- The PC contains a natural number, which is the memory address where the next Z-code instruction is to be found. Note that this is not limited to the size of a word or longword. This means that in principle memory can have any size (although there are reasons that limit the size of memory in practice).
- The routine stack is a stack of words, which is unlimited in size. A word can be pushed on top of the stack, or pulled off again. It is an error to pull a word from an empty routine stack. A routine stack can only be used to store values within one (execution of a) routine; in particular, it cannot be used to pass values from one routine to another.
- The local variables contain words. There are 0 to 15 local variables (numbered from 1 onwards), and this number can not be changed. It is an error to refer to a non-existing local variable.
- In V3+, each frame also stores the way this routine was called: as a function, procedure, or interrupt. (This is used on a `return` instruction; see 8.5.)
- In V5+, each frame also contains the number of values passed with the call that created this frame. (This is used to implement the `check_arg_count` instruction; see 8.5.)

It is an error for any instruction to leave the call stack empty. Because of the way the Z-machine works, this amounts to saying that it is an error for a return instruction to be encountered in the 'main routine.'

In V5+ a frame on the call stack can be tagged with a word, called the frame pointer; at any time, all frame pointers of the call stack must be different. A frame pointer is created by the `catch` instruction, and removed when its frame is removed from the call stack. Frame pointers are used in the instructions `catch` and `throw`.

The call stack has a link to a 'save file' outside the Z-machine. Through this link the current contents of the call stack can be saved, and read back later (see `save` and `restore` in 8.13).

> *Note:* Although in theory the call and routine stacks are unlimited, most emulators will have to limit their lengths. As a minimum the present limit of ZIP is recommended, which is as follows. With each frame we associate a 'size' equal to the number of local variables in it, plus the length of its routine stack, plus 4. The emulator should then operate as specified as long as the total 'size' of the frames on the call stack remains less than 1020.

2.3. Memory

The memory consists of an unlimited number of bytes, numbered by addresses from 0 onwards. We will refer to a sequence of bytes (byte, word, etc.) stored from address *n* upwards as "the … at *n*".

Z-programs use a number of ways to refer to addresses; each is represented as a word. A byte address is an unsigned word that is simply read as an address. A word address is an unsigned word *w* that is read as the address $2 * w$. (Note that word addresses are only used in the abbreviations table; see 3.2.) In order to be able to reference even higher addresses, the notion of a packed address is introduced. This is a word *w* that is read as an address *a* using the following formula:

$$a = \begin{cases} 2 * w & \text{for V } 1 - 3 \\ 4 * w & \text{for V } 4 - 5 \\ 4 * w + 8 * o & \text{for V } 6 - 7 \\ 8 * w & \text{for V } 8 \end{cases}$$

here, the offset *o* is the contents of the header word at $28 (for routines) or at $2A (for Z-strings). Note that this means that (in V6-7) there are two different kinds of packed addresses, one for routines and one for Z-strings.

On a signal from the CPU, a part of the memory beginning from address 0 is initialized with a Z-program via a link to the outside world. The length of a Z-program is the number of locations that are initialized. It is an error to reference a non-initialised byte of memory. From now on, when we speak of ‘memory,’ we mean ‘the initialized part of memory.’

Although not strictly necessary, the following limit is imposed on the length of a Z-program:

V1-3:	128K
V4-5:	256K
V6-7:	576K
V8:	512K

It is unspecified what happens when a Z-program is run that exceeds this limit. It is recommended that it is processed as usual.

Depending on the contents of the memory, its locations are divided into RAM, ROM, and IROM (for Initialisable ROM). The contents of every location can be read, but only RAM may be written to by Z-instructions. It is an error to write to ROM. IROM may be written to by the CPU on special occasions (*e.g.*, on start-up), but it is an error for a Z-instruction to directly write to IROM.

The first 64 bytes of memory are called the header; this is a mix of RAM, ROM and IROM. The meaning of the header information, and the types of its locations, are laid out in table 3 in the appendix.

Note: To help disassemblers, it is strongly recommended that all unused bits and bytes of a Z-program are set to 0. Note that this is mandatory for the header; see table 3 in the appendix.

Memory is divided into two contiguous parts, which are known as dynamic and static memory, respectively. (Dynamic memory is also known as the ‘save area.’) The byte address in the header word at $0E gives the address of the first location of static memory. It is an error if there are non-initialized locations before this one. Every non-header byte in dynamic memory is RAM; static memory consists entirely of ROM.

It is unspecified what happens if dynamic memory is less than 64 bytes long. Recommended behaviour is to behave as described, or else to display an error message and halt.

Note: Because of the Z-machine’s instruction set, above address $FFFF only strings and routines stored at packed addresses can be accessed by a Z-program. This means that at most 64K is available for general data.

The memory, just as the call stack, has a link to the outside world that is used to read or write the contents of dynamic memory. In V5+, there is a similar link to the save memory.

Finally, there is another way to divide memory into two contiguous parts: resident and paged (or high) memory. This distinction has no effect on the behaviour of the Z-machine at all, but can be used to speed up its emulation. It was made to help emulators on computer systems that do not have enough memory to store an entire Z-program. The resident part of Z-machine memory should be quickly accessible, for speed reasons; in practice, this means that this part should always be in computer memory. For the paged part, speed is not that critical; in practice, pieces of paged memory (‘pages’) can be read from disk when they are needed.

Paged memory begins at the byte address stored in the header word at $04. It is unspecified what happens when paged memory begins in dynamic memory; it is recommended that paged dynamic memory can be read and written just as resident dynamic memory. Note that most existing emulators can not handle this situation.

2.4. I/O Card

The I/O card, which is connected to the CPU, handles all character based input and output of the Z-machine. It reads input from the keyboard (and in V6 the optional mouse), or from a command script; it can output to the video card, a game transcript, a command script, and memory. For this reason it has access to the different fonts and styles. A detailed description of its operations is deferred to the section 6.

2.5. Video card

The video card is placed between the I/O card and the screen, and its job is to show the text output of a Z-program on the screen in windows. In V6 it also has access to data for showing pictures; these are not stored anywhere in Z-machine memory, and the Z-machine neither knows nor cares about the picture format that is used.

The Z-machine neither knows nor cares about the format of the picture data. Pictures are just rectangles of units, and they are numbered with non-zero natural numbers. The video card answers questions from the CPU about the height and width of the pictures it has access to. A V6 Z-machine should set header bit $10-$11/3 correctly on start-up, to let the Z-program know whether it can show pictures or not. It is unspecified what happens when a V6 machine without picture capability encounters a picture instruction. Recommended behaviour is to ignore the instruction.

The capabilities of the video card are described in detail in section 4. Because the video card always handles one screen, in the rest of this document we will often ignore the distinction between these two. Note however that the screen is not a part of the Z-machine, but just a rather dumb output device that only knows how to show rectangles of (possibly coloured) characters or pixels, and (in V6) can be asked about the displayed colours.

2.6. Random number generator

The random number generator is used to introduce a chance element in Z-programs. On request, it generates a number in the range from 1 to *n* inclusive, where *n* is a given number between 1 and 32767 inclusive; see the `random` instruction in 8.14. The generator is in one of two modes: predictable or unpredictable. Initially, it is in unpredictable mode, which causes the generated numbers to be really random and uniformly distributed. When putting the generator in predictable mode, a ‘seed’ is supplied. In this case, for the same seed the same sequence of requests is guaranteed to generate the same sequence of numbers.

> *Note:* Z-programs should not rely too much on the randomness of the generated numbers. Emulators should try hard to produce good random numbers; there is a case known where a bad generator made a game unwinnable.

2.7. Sound Card

From V3 the Z-machine contains a sound card. This card has access to the sound data; these are not stored anywhere in Z-machine memory, and the Z-machine neither knows nor cares about the sound format that is used. A Z-program refers to sounds by numbers beginning from 1. Sounds 1 and 2 are a high-pitched and a low-pitched beep respectively. The other sounds (‘sound effects’) are unspecified, and are generally emulator-dependent.

Header bits $10-$11/4 (V3), $10-$11/7 (V5+), and $01/5 (V6) indicate whether the sound card can produce sound effects. These bits should be set on start-up, according to the configuration of the Z-machine. The sound card should produce the sound effect that is asked for, or a good substitute; if that is not possible the sound card should be silent, which in most cases has the same effect as ignoring the sound instruction. (If a sound effect cannot be played, a warning message might be displayed by the emulator.)

The sound card plays at most one sound effect at any given time, independently of the operation of other parts of the Z-machine. A sound effect can be played at 8 volume levels, it can (in V4+) be repeated 1 to 254 times or indefinitely, and (in V5+) an interrupt can be generated when playing has been completed. Beeps are played independently of sound effects. Initially the sound card is silent.

2.8. Timer

From V4, a timer is added to the Z-machine. This makes it possible for the CPU to wait for a given number of (real world) seconds in the range of 0.1 to 6553.5 seconds, with a resolution of 0.1 seconds.

2.9. Save memory

In V5+ the Z-machine optionally has a save memory, which can be used to store and restore a snapshot of the dynamic memory and the call stack of the Z-machine. On a signal from the CPU, the contents of dynamic memory and call stack are copied to save memory, or vice versa. It is unspecified what the contents of the save memory are directly after start-up, a restart, or a `restore_undo` instruction; a `restore_undo` instruction fails in these cases.

[**TODO**: Add something on multiple undo?]

2.10. Mouse

A V6 Z-machine can optionally be connected to a mouse; if none is present at start-up, header bit $10-$11/5 should be cleared. Mouse clicks are reported to a Z-program as special input characters, the so-called ‘function keys’; see section 5.1. On a mouse click, the screen coordinates of the mouse pointer are written in the first two words of the ‘extension table,’ (see 3.9). If the Z-machine is connected to a mouse, this table must have at least two entries. The first of these contains the mouse x coordinate, the second the y coordinate.

It is possible to constrain the mouse pointer to a given screen window, using the `mouse_window` instruction; see 8.12. In this case all clicks outside this window are ignored. Initially the mouse pointer is constrained to window 1.

2.11. Initialization

When the Z-machine is started or restarted, a number of initialisation steps are performed.

- On restart only, the current value of the ‘printer transcript bit’ (header bit $10-$11/0) is remembered.
- The memory is initialized with a Z-program; on a restart, this is the same Z-program as used on start-up.
- A number of bits and bytes in the header are written, depending on the capabilities of this specific Z-machine. This includes most of the bits marked “IROM” in table 3 in the appendix. On restart only, the ‘printer transcript bit’ is set the to the value remembered in the first step.
- All components are initialized; *e.g.*, the screen is cleared, the sound card is silenced, etc.
- The call stack is made to contain a single frame, containing in V1-5:
  - a PC which is set to the byte address in the header word at $06;
  - no local variables;
  - an empty routine stack.

  In V6 execution begins at a routine, the packed routine address of which is in the header word at $06. The single start-up frame is now almost the same as the one created for a call instruction using that address:
  - a PC which is set to that address plus 1;
  - a number of local variables which is found in the byte at that address, all initialized to zero;
  - an empty routine stack.

  In V5+ it is unspecified what the number of passed values to this routine is set to; 0 is the recommended value. Note that in V5+ it is not necessary to specify how this routine was called, as this fact is never used: a return instruction is illegal from the 'main routine.'

After this, the Z-machine begins to execute Z-code.

2.12. Operation

While the Z-machine is running, the following steps are repeated until it is halted (*i.e.*, a `quit` instruction is carried out):

- If an interrupt has been generated by an outside event (*i.e.*, a sound effect has finished), that routine is called as an interrupt without passing it any values, and ignoring the returned value. This step is then executed again.
- The Z-code instruction beginning at the PC address is read. (This excludes the so-called string, result, and branch arguments.) Note that it is legal for a Z-code instruction to appear in RAM; this makes self-modifying code possible.
- The PC is set to the address after the instruction.
- The instruction is carried out, possibly resulting in changes to the PC, or any other writable locations.

It is an error to try to execute an instruction that is not recognized by the Z-machine which is used. Still, after displaying a suitable message, an emulator might try to ignore the offending instruction and continue execution. This is especially the case with unrecognized 'extended instructions' (see section 7.1) which are probably post-Infocom extensions.

3. Data Structures

There are a number of regions in a Z-program containing information that is used in a special way by the Z-machine. This section describes these data structures, beginning with those concerning text.

3.1. Z-strings and Z-characters

A Z-string is a sequence of words that represents a piece of text. Text is in fact just a sequence of output character numbers (see 5.1) to be printed in the current font and style.

Each word of a Z-string consists of an end marker (bit 15) and three Z-characters of five bits each (bits 14 to 10, bits 9 to 5, and bits 4 to 0). Only the last word of a Z-string has an end marker of %1. In this way a Z-string is in fact a sequence of Z-characters. (Note that many Z-strings stored in memory begin at packed string addresses, but this is not mandatory.)

Note that the concepts 'Z-character' and 'output character' must not be confused. Also note that the term 'word' is used exclusively for a sequence of 16 bits.

A Z-character is a number in the range of 0 to 31. The meaning of a Z-character depends on the current alphabet. There are three alphabets, for lower case (L), upper case (U) and punctuation (P). The following table shows the built-in character set. A circumflex (^) indicates that this Z-character has a special meaning that is described below. The other Z-characters are converted to the ASCII-codes of the characters shown.

```
              1    1    2    2    3
    0    5    0    5    0    5    0
L    ^^^^^abcdefghijklmnopqrstuvwxyz
U    ^^^^^ABCDEFGHIJKLMNOPQRSTUVWXYZ
P    ^^^^^^0123456789.,!?_#’"/\<-:()   (V1)
P    ^^^^^^^0123456789.,!?_#’"/\-:()   (V2+)
```

As an example, Z-character 6 in alphabet L represents the output character 97, which is the ASCII code for ‘a’. Note that Z-character 0 means output character 32 (an ASCII space) in all alphabets.

Note that the distinction between the character shown in the above table and its ASCII-code becomes important when using a non-ASCII font (see 5.2).

The special codes have the following meaning. In V1-2, characters 2 to 5 are used for alphabet changes. In V1, character 1 means newline in all alphabets. In V2+, character 7 in alphabet P means newline. In V2, character 1 is an abbreviation character. In V3+, characters 1 to 3 are abbreviation characters, and only characters 4 and 5 are used for alphabet changes. In all versions, character 6 of alphabet P indicates that a literal output character is given.

Because the number of characters in a Z-string is always a multiple of 3, in some cases extra characters have to be added – this is called ‘padding.’ For this purpose all alphabet change characters can be used; conventionally character 5 is used, as this is also the pad character used by `encode_text`. The use of character 5 is mandatory in padding entries in a sorted dictionary (see 3.3).

In V5+ a Z-program can contain an alternative character set, which replaces (part of) the built-in one described above. The word at $34 in the header contains either 0, or the byte address where the alternative character set is stored. It consists of three sequences of 26 bytes, for Z-characters 6 to 31 from alphabets L, U and P, respectively; each of these bytes is interpreted as an output character. Characters 0 to 5, and characters 6 and 7 from alphabet P, have the same meaning as for the built-in character set. (This means that the alternative values for characters 6 and 7 from alphabet P are ignored.) It strongly recommended that all output characters in an alternative character set are different. [Question: Should this be mandatory?]

3.2. Converting a Z-string

This subsection describes how to convert a Z-string to a sequence of output characters. This is done by taking one (or sometimes two or three) Z-characters at a time. During conversion two alphabets are remembered, called the ‘current’ and ‘lock’ alphabets. The current alphabet is the one in which the next Z-character must be interpreted, while the lock alphabet gives the long-term interpretation. After converting one (or two or three) Z-characters that was not an alphabet change, the current alphabet is set to the lock alphabet. At the beginning of converting a Z-string both alphabets are L. (Note that the description below implies that in V3+ the lock alphabet is always L.) The next paragraphs state how to convert individual Z-characters: ordinary characters, newlines, abbreviations, literal output characters and alphabet changes.

A Z-character that is not a special code in the current alphabet is converted to an output character, using either the built-in or the alternative character set; see above.

A newline Z-character (*i.e.*, 1 in all alphabets in V1, or 7 in alphabet P in V2+) is simply converted to output character 13, *i.e.*, a newline.

An abbreviation character (*i.e.*, 1 in V2, and also 2 and 3 in V3+, in all alphabets) is always interpreted together with the next character. These characters represent an abbreviation, which is stored as a Z-string. (Incomplete abbreviations are ignored.) If *a* is the abbreviation character and *b* is the next character, then these point to abbreviation Eq(a-1)*32+b. A table of abbreviations is stored in memory (usually in RAM) beginning at the byte address stored in the header word at $18. This is a contiguous list of 32 (in V2) or 96 (in V3+) words, which are the word addresses where the abbreviation Z-strings are stored. An abbreviation is converted to whatever its Z-string converts to. It is unspecified what happens if an abbreviation Z-string itself uses abbreviations; recommended behaviour is to behave as described above, and give an error message if abbreviations are used recursively. [Question: [Nelson] says that an abbreviation Z-string may not end with an incomplete multi-Z-character construction. I don't see why not.]

Character 6 in alphabet P is always interpreted with the next two Z-characters, and converted to a single output character of 10 bits. (Incomplete literal output character constructions are ignored.) This is done by taking the five bits of the first of these characters, followed by the five bits of the second.

Alphabet changes are converted to nothing, but they can change the current and lock alphabets. Their effects are given in the following table:

char.	L	U	P
2,4	U	P	L
3,5	P	L	U

(Note that only 4 and 5 are alphabet changes in V3+.) This gives for the current alphabet and for each character from 2 to 5 the alphabet that is changed to, *i.e.*, the new current alphabet. If the character is 4 or 5, in V1-2 the lock alphabet is also set to this alphabet; otherwise it remains unchanged. Note that this means that the lock alphabet is always L under V3+. Note also that a sequence of alphabet changes at the end of a Z-string has no effect.

> *Note:* Because of differences between existing emulators, it is strongly recommended that the writer of Z-strings avoids multiple 'shift' alphabet changes (V1-2: 2,3; V3+: 4,5) in a row, except for padding purposes (see above). Such a sequence is otherwise useless, since it can be replaced with a single alphabet change having the same effect.

[Question: For V3+ the behaviour described here differs a bit from what [ZIP] and [Frotz] do and [Nelson] says, but it is what Paul David Doherty says Infocom interpreters do. In effect, in V3+ ZIP and [Nelson] set the current alphabet to L before processing an alphabet character in the way described above. This difference shows itself in some unlikely cases. An example is a Z-string beginning with Z-characters 5, 5, 20. The described procedure converts this to 'O' (character 20 from the U alphabet), but ZIP gives '!' (from the P alphabet). On a related note, the ITF interpreter generates '!' for this sequence in V3+, but all following characters are interpreted in the P alphabet too. This means it erroneously has a 'shift lock' in V3+.]

3.3. Dictionaries

The main dictionary, which is pointed to by the byte address in the header word at $08, begins with a header. This consists of the separators (a 1-1-table of input characters), an (unsigned) byte representing the length of each entry, and a (signed) word representing the number of entries that follow the header. Then follow the entries themselves.

It is an error for a dictionary to have an entry length of less than 4 (V1-3) or 6 (V4+). The first 4 (V1-3) or 6 (V4+) bytes of an entry contain a Z-string (of 6 or 9 Z-characters), and the Z-machine does not care about the contents of the other bytes. [Question: Should we demand that the Z-strings are unique? Or perhaps only in the sorted case?] To be useful, the Z-string in an entry must not use abbreviations or characters 6-31 from alphabet U, and use 5 as the padding character; for an explanation, see `encode_text` in 8.14.

The dictionary entries must be sorted on the Z-strings they contain, except as noted below. The Z-strings are compared in numerical order, regarding their 4 or 6 bytes as an unsigned integer. This means that the order is more or less alphabetical for the built-in character set.

In V5+ there are instructions that can use an alternative dictionary which has the same structure as the main one. In an alternative dictionary, however, it is legal to give a negative number of entries, say $-s$. This means that there are s entries, but that they are unsorted. It is an error for a dictionary with a positive number of entries to be unsorted. It is unspecified what happens when the main dictionary is unsorted. Recommended behaviour is that it is used as usual.

3.4. Objects, their attributes and properties

Simulation of a world using the Z-machine relies on objects. An object stores all relevant information about a single entity in the simulated world. The number of objects is finite, but is not stored explictitly in a Z-program. Objects can be changed (and even added or deleted) by writing directly to the memory locations where they are stored – but this is not recommended.

Every object has a name, attributes, and properties. There is a fixed set of attributes. For each object, each attribute is either *true* or *false.* There is a fixed set of properties. For each object, a property is either present on the object, and contains a sequence of data bytes; or it is absent. For every property, there is a word that gives a default value; this is used when reading a property which is not present on an object.

The objects are arranged as a collection of family trees. Every object has at most one parent, and zero or more children (which are thus siblings of each other). Each object knows only three of its relations: its parent, its first child, and its next sibling; or commonly: its parent, child, and sibling.

The object tree must be consistent, *i.e.*, the following must hold:

- Every object that has a (next) sibling also has a parent.
- Each object is the parent of the objects in the ‘sibling chain’ that begins with its (first) child, and only of those. (This chain is understood to be empty if there is no (first) child; the object is then the parent of no object at all.)
- No object occurs in the ‘sibling chain’ that begins with its own (next) sibling. (As above,

this chain is empty if the object has no (next) sibling.)

It is an error to use an inconsistent object tree, but most emulators are probably not able to check this properly. Note that the object tree is consistent if it is so initially, and only the `remove_obj` and `insert_obj` instructions are used to change it. The emulator might show a warning message if the object tree is initially inconsistent.

Note that recursive trees are not forbidden, because there exist Infocom games that produce them (although due to bugs). They can be avoided by following this additional consistency rule:

- No object occurs in the 'parent chain' that begins with its own parent. (As above, this chain is empty if the object has no parent.)

This can be ensured if the object tree is initially consistent, and the `insert_obj` instruction is used carefully; see its description in section 8.

Note that Inform [Inform] uses an 'aged' terminology: the first child of an object is called the eldest, and the last of its children the youngest. This is a bit odd, since `insert_obj` – the only instruction to add children – adds an object as the first of the sibling chain; it seems reasonable to refer to this newly added object as the *youngest* child.

3.5. Representing objects

Because the representation of objects differs between V1-3 and V4+, this subsection gives a description for V1-3, with changes for V4+ put in parentheses.

Objects are numbered from 1, and 'object number' 0 is often used to mean 'no such object exists.' Note that this is sometimes called 'object 0.' That is not done here, since this 'object' is unlike all others: it has no name, no attributes, and no properties.

The number of objects is at most 255, an unsigned byte (65535, an unsigned word). [Question: Frotz 2.01 uses a maximum of 2000 objects. Where does this restriction come from?] It is an error to reference a non-existing object, but most emulators do not check this. Attributes are numbered from 0 to 31 (47). Properties are numbered from 1 to 31 (63). Each property present on an object contains 1 to 8 (63) data bytes; this length can not be changed during execution of a Z-program, except by directly writing the property lists – which is legal, but not recommended.

The byte address in the header word at $0A is either 0, and there are no objects; or it points to the beginning of the object table. The object table consists of two parts: first comes the property defaults table, and then the object entries. Each object entry stores its attributes and relations, and a pointer to its property list; this stores the property information for the object, and also its name.

If the Z-program has objects, it is mandatory that a property list be stored directly after the last object entry. This requirement makes it possible to compute the number of objects. [Question: Why should this be mandatory, and not a recommendation? Failing this condition would simply mean that tools like 'txd' cannot compute the number of objects.] Note that existing Infocom and Inform game files store all property lists there, and Inform even puts them in object order.

Note: In theory the number of objects is *not* fixed, since the memory locations where they are stored can be arbitrarily manipulated by the Z-program. In practice most emulators assume that this is not done, but a really friendly emulator might issue a warning message if it is.

The property defaults table contains 31 (63) words, giving the default property data for each property. It is followed by the object entries, beginning from object 1. Each entry contains 4 (6) attribute bytes, followed by three bytes (words) indicating the parent, sibling, and child objects, and finally a word giving the byte address at which the object's property list begins. The top bit of the attribute bytes refers to attribute 0, the bottom bit to attribute 31 (47). If an object number in an entry is 0, this means that the object has no such relation.

A property list consists of this object's name – which is a Z-string preceded by an unsigned byte containing its length in words – a number of property entries, and a zero byte that terminates the list. A property entry contains one (one or two) size byte(s), followed by 1 to 8 (63) bytes of property data; the (first) size byte holds the number of data bytes minus 1 in the top 3 (2) bits, and the property number in the bottom 5 (6) bits. (In V4+ there is one exception: if the top bit of the first size byte is %1, *i.e.*, the number of bytes would be either 3 or 4, there are two size bytes. The second size byte contains the number of data bytes as its bottom 6 bits, and its top bit is %1; the number of data bytes may not be 0. The use of bit 6 of a second size byte is unknown; Inform always sets it to %0.) The entries in a property list must be sorted in order of descending property number; it is an error for a property number to occur more than once in a property list. It is also an error for property number 0 to occur.

3.6. Global variables

There are 240 global variables, numbered 0 to 239. As with the local variables, each global variable contains a word. Unlike the local variables, the global variables are stored in memory (usually in RAM), as 240 consecutive words beginning from the byte address found in the word at $0C.

For V1-3 the first three global variables have a special meaning in constructing the status bar; see section 4.

Note: In V1-3 global variable 0 contains the number of the object representing the location where the player is. It is strongly recommended that global 0 is always used in this way, if the Z-program has a notion of 'current location.' Similarly, if a score and a number of turns (or a time in hours and minutes) is kept, it is strongly recommended to store these in globals 1 and 2 respectively.

3.7. Routines

A routine is nothing more than a very simple header that is put before Z-code instructions, and always begins at a packed routine address. The first (unsigned) byte of a routine indicates the number of local variables the routine has. It is an error for this to be greater than 15. In V1-4, this is followed by the same number of words, giving the default initial values for the local variables. (Note that in V5+ a default initial value of zero is assumed.) Execution of the routine begins at the address after this routine header.

A routine can be called in three different ways: as a function or procedure, using a `call` instruction; and as an interrupt, on specific events, such as an input timing out, a sound effect finishing, and a ‘newline count’ reaching zero. Details about `call`ing and `return`ing from routines can be found in 2.2 and 8.5. When a routine is called as an interrupt, in effect a `call` instruction is executed, but the newly created call stack frame now stores that this was an interrupt call.

[Question: A non-Infocom extension that could easily be implemented using interrupts is an action to be taken when the screen size changes. A word in the extension table (see 3.9) could be used to point to an interrupt routine to be called whenever the screen size is changed, zero indicating that no interrupt routine is provided. If there are plans to introduce more interrupts, it might be better to use an interrupt table, pointed to from the extension table.]

3.8. User stacks

User stacks are available in V6. A user stack is a stack of words that, unlike the routine stack, is stored in memory. It consists of an unsigned word representing the number of words that can still be pushed onto it (the stack pointer), followed by that many words (the free words), and after that the words that are on the stack (beginning with the top element).

When the stack pointer is 0, it is impossible to push anything onto the stack. When a word is pushed onto a user stack, it is written in the highest free word, and the stack pointer is decremented by one. When a word is pulled off, the stack pointer is incremented by one, and the contents of the (now) highest free word is returned.

Note that there is no way to detect underflow (*i.e.*, pulling off an empty stack) for user stacks: a user stack is a bottomless pit. A Z-program must make its own arrangements to remember where the bottom of the stack is located.

3.9. Extension table

The extension table is a 2-2-table beginning at the byte address stored in the header word at $36. (If there is no extension table, this address is 0.) The first two words, if present, store the x and y coordinates of the mouse pointer (see 2.10). The meaning of the other entries, if present, is unspecified; they might be used for future Z-machine extensions.

4. The Video Card

To make interaction possible there must be a way to put data into the Z-machine, and get results out of it. The center of interactivity is the screen, which is only accessible via the video card. This section describes the operations that the video card can perform, and its initial state.

4.1. The screen

The screen that is connected to the video card is a rectangle of units, a unit being the smallest addressable part of the screen. A unit is always indicated by its position relative to some origin, as a pair (y, x). y gives the number of units minus 1 to do down from the origin; x gives the number of columns minus 1 to go to the right. This means that the origin itself has coordinates (1,1), relative to itself. All dimensions and distances on the screen are expressed in units, unless otherwise stated.

Note that a unit usually contains a character or a pixel, but this is not specified. [Question:

Nelson [Nelson, 8.4.2] says that a unit must contain a character in V1-4. This seems incorrect, as it would make an emulator using a variable-width font impossible for these versions.]

The size of the screen is set at start-up, and can be changed at any time. It is unspecified what the screen looks like after a screen size change; it is recommended that the screen is cleared (using an EraseScreen operation). The screen can be set to be infinitely high; in this case it has no bottom edge. [**TODO**: Add how the header bytes at $20 and $22 (in V4+), and the header words at $24 and $26 (in V5+), should be set.] [Question: What does 'number of lines' mean when a V4+ interpreter uses styles of different heights? And what does 'number of characters per line' mean when a V4+ interpreter uses a variable-width style?] [Question: Is an infinite height legal in all versions? What should the 'screen height in units' (header word $24) be in this case in V5+? Or is that value ignored?]

The contents of the screen can only be written, not read, with one exception: at any time the foreground colour of the unit at the current window's cursor position (see below) can be read. This is used by the `set_colour` instruction. The screen contents can only be changed by using the operations of the video card.

4.2. Windows

The video card is used to access the screen in an orderly fashion. For this purpose it provides a number of windows to which output can be sent. The state of the video card is completely determined by the properties of the windows, the current window, and the state of the screen cursor.

The number of windows is fixed: it is 1 in V1-2, 2 in V3-5, and 8 in V6; they are numbered from 0 upwards. A window is a rectangle of screen units, with a location determined by its top left corner and its vertical and horizontal size. To address a unit within a window, the top left corner is used as the origin (1,1).

For every window the video card keeps an number of properties. Changing these doesn't change the contents of the screen; to do that, an output operation must be performed. The properties include location and size, cursor position, buffer mode and other attributes, font, style within that font, colours, etc. In principle each of these properties can be changed separately.

Note: The attributes and properties of windows should not be confused with those of objects.

At any given time there is one window that is used for output operations; this is called the 'current' or 'selected' window.

At any given time either a 'screen cursor' is shown at the cursor position of the current window, or there isn't.

The following subsections describe the properties of the windows, the operations of the video card, and its initial state.

4.3. Window properties

Each window has the following properties, which are all unsigned words:

0. Y location
1. X location

These give the position of the top left corner of the window, in units relative to the top left of the screen. It is legal for this position to be outside the screen. Only output to the visible part of the window is shown; the rest is ignored.

2. Y size
3. X size

   These give the height and width of the window in units. Again, it is legal if part of the window is off the screen; see above.
4. Y cursor
5. X cursor

   These give the position of this window's cursor, in units relative to its top left corner. It is legal for the cursor to be outside the window, but it is illegal to try to output something to it in that case.
6. Left margin size
7. Right margin size

   These give horizontal margins that the text should be kept between, in units from the window edge inwards. A value of 0 means that the margin coincides with the window edge.
8. Newline routine (packed routine address)
9. Newline countdown

   See the description of the NewLine operation below.
10. Style within current font

    The number of the style for the next output.
11. Foreground (first byte) and background (second byte) colours

    The meaning of these colour numbers are:

    2. black
    3. red
    4. green
    5. yellow
    6. blue
    7. magenta
    8. cyan
    9. white
12. Font

    The number of the font to be used for the next output.
13. Style height (first byte) and width (second byte)

    The height of the style indicated by properties 10 and 12. This is the only property that is not writable; it changes with properties 10 and 12.
14. Attributes

    The bits of this word contain a number of attributes of the window; these are detailed below.
15. Line count

    [Question: What is the meaning of this property? How and when should it be changed? Note that e.g. 'Zork Zero' only sets this to -999.]

Property 14 of each window contains 16 attributes (numbered 0-15), which can be either on or off. These are stored in the bits with the corresponding number, where %1 means that the attribute is on. Only the meaning of window attributes 0 to 3 is specified:

0. Is character wrapping on?
   [Question: What is the exact meaning of this attribute?]
1. Is scrolling on?
   This indicates whether it is allowed to scroll the contents of the window up to make room for a new line.
2. Is output to this window transcripted?
   This controls whether output to this window is to be sent to output stream 2.
3. Is the buffer mode on?
   This bit indicates whether output to this window is sent directly to the screen (buffer mode off) or is first buffered to avoid consecutive non-spaces being spread over two lines.

Only under V6 can all properties (except 13) be read and written. For earlier versions, some are either ignored or cannot be changed during execution.

4.4. Operations

This subsection describes the output operations that can be performed on the video card. Only by using these operations the contents of the screen can be changed.

- ShowStatusBar(*s*, *a*, *b*, *flag*) — Show the status bar with the given data. (Only used in V1-3.)

  The top lines of the screen are filled with a status bar containing the given data. *s* is a sequence of output characters without newlines; this represents the name of the current location. *a* and *b* are signed numbers representing either score and number of turns (if flag is %0) or the time in hours and minutes (if %1).

  The height of the status bar is unspecified, but must remain constant during execution of a Z-program. Note that this height might be 0 units, in which case no status bar is shown; it is also legal for the status bar to occupy more than one line, to accommodate narrow screens.

  The contents of the status bar are unspecified, but it show as much of the given data as possible. The following describes the recommended format. It consists of one line, and is printed in the inverse video style of the roman font (see section 5.2). On the left hand side appears the name of the current location. If the name cannot be completely shown, an initial part should be shown with an ellipsis ("…") added. If flag is %0, on the right hand side appears "Score:", a , "Turns:", b ; if %1, "Time:", a , ":", b . The rest of the status bar is empty. It is allowed to display time in an a.m./p.m. notation (as ZIP and Frotz do) since Infocom did this too. However, this gives strange results for V3 Z-programs not set on Earth. [**TODO**: Make this description more precise, and state 23 character limit for location name. State that this is Infocom behaviour.]

  Note that this operation is used at specific moments only, viz. while executing a `read` or `show_status` instruction.

- ScrollWindow(*w*, *y*) — Scroll the contents of the given window up over the given distance, filling with the background colour and moving the cursor also.

  The contents of window *w* are moved *y* units up, thus moving down if *y* is negative. The units that scroll away are overwritten, the 'new' part is filled with the background colour.

The cursor is moved the same way by adding *y* to the vertical cursor position.

> *Note:* Note that the ITF emulator and some versions of ZIP erroneously fill with the foreground colour if an inverse video style is selected for the window.

- ShowChar(*w*) — Show the given character description (with colours) at the cursor position of the current window.

  (See section 5 for more information on character descriptions.) Call the height of the description (in units) *h* and its width *w*. It is an error if the character doesn't fit between the cursor position and the right window edge. If it doesn't fit between the cursor position and the bottom window edge, and if the scrolling attribute is set, a ScrollWindow(*n*,*m*) operation is performed, where *n* is the current window, and *m* the number of units needed to fit the description in. It is an error if the character still doesn't fit.

  The character description is now written to the screen with the given colours, taking the cursor position as the upper left corner. Then the cursor is moved to the right by *w*.

- NewLine — In the current window, move the cursor to the left margin of the next line.

  Move the cursor down the height of the style of the current window, and to its left margin. If its newline countdown property is not zero, decrement it by one; if it then becomes zero, call the newline routine as an interrupt without passing it any values, and ignoring the return value.

  Note that scrolling is handled by ShowChar instead of NewLine. This is done to handle text lines with fonts of different heights. Scrolling on a newline would scroll up by the height of the then current style, but if higher characters follow an error would result. [**TODO**: Change if all fonts have the same height.]

  Note that strictly speaking NewLine is not an output operation, since it doesn't alter the contents of the screen. It was made into a separate operation because of its complexity.

- ShowPicture(*p*, *y*, *x*) — Show the given picture in the current window.

  Show picture *p* in the current window with (y, x) as the top left corner. [Question: What happens when the picture doesn't fit?]

- ErasePicture(*p*, *y*, *x*) — Erase a rectangle of the size of the given picture in the current window.

  In the current window, fill a rectangle of the size of picture *p* with (y,x) as the top left corner with the background colour. [Question: See ShowPicture.]

- EraseWindow(*w*) — Erase the given window.

  Fill the given window with its background colour. In V1-4 the cursor is moved to the bottom left of the window, in V5+ to the top left.

- EraseScreen — Erase the entire screen.

  Fill the entire screen with the default background colour, which is found in the header byte at $2C in V5+ and unspecified in V1-4. Nothing else happens.

4.5. Initial state

When the video card is initialized (*i.e.*, on start-up or restart), the screen is cleared, all window attributes and properties are set to their initial values (see below), the buffer is cleared, `ExtraNL` (see below) is set to *false*, window 0 is selected, and the screen cursor is turned on.

The following table gives the initial values of the attributes and properties for all versions, and also states whether they can be changed during execution of a Z-program.

Table 1. Initial values for properties and attributes

		V1-2	V3	V4	V5	V6
prop	0-5	D *	D **	D **	D **	D
	6	0 *	0 *	0 *	0 *	0
	7	0 *	0 *	0 *	0 *	0
	8	—	—	—	—	?
	9	0 *	0 *	0 *	0 *	0
	10	0 *	0 *	0 **	0 **	0
	11	U *	U *	U *	H **	H
	12	1 *	1 *	1 *	1 **	1
	13	U *	U *	U *	U **	U
	15					
attr	1	0 *	0 *	0 *	0 *	0
	2	1 *	D *	D *	D *	D
	3	1 *	D *	D *	D *	D
	4	1 *	1 *	1 **	D	1

Meaning of the symbols:

—:	not used;
?:	no default value;
D:	different for different windows;
H:	a default value from the header is used;
U:	unspecified, i.e., emulator-dependent;
*:	cannot be changed using Z-code instructions;
**:	changes for all windows at the same time.

In all versions, initially window 0 is the only window that has the scroll and transcript attributes set. For V1-5 this cannot be changed, but for V6 these can be toggled.

Buffer mode is always on for all windows in V1-3. In V4 it is initially on for both windows, and is toggled for both at the same time. In V5 it is initially on and can be toggled for window 0, and always off for window 1. In V6 buffering is on by default for all windows. [Question: Is this correct?]

In V1-4, and in V5+ when header bit $01/0 or $10-$11/6 is cleared, the colour of characters on the screen is unspecified and cannot be changed. In all other cases the initial colours are found in the header bytes at $2C and $2D. In V5 these can only be changed for both windows simultaneously, but in V6 each window has its own colours.

In V1-2 there is one window, which occupies the entire width of the screen. The lines of the screen are divided into two regions: the status bar (see below) and the window. The dimensions of the window cannot be changed.

In V3-5 there are two windows; window 0 is known as the ‘lower’ and window 1 as the ‘upper’ window. Both occupy the entire width of the screen. The lines of the screen are now divided into two or three regions: the status bar (if present; see below), the upper window, and the lower window. The upper window initially has zero height, and this can only be changed in V3+ using the `split_screen` instruction. Note that if header bit $01/5 is cleared in V3, the `split_screen` instruction is illegal, or in other words: the upper window is never shown.

The status bar is only present in V1-2, and in V3 when header bit $01/4 is cleared. Its height is unspecified, but constant and non-zero (see the ShowStatusBar operation above for more details).

A V6 Z-machine has 8 windows, which can each be located anywhere on the screen. It is legal for a window to extend beyond the screen: everything written outside the screen is simply ignored. Initially all windows have their top left corners at the top left corner of the screen; window 0 occupies the entire screen, window 1 spans the width of the screen and has zero height, and all other windows have zero height and width. A status bar is never shown.

The initial cursor position for a window is its top left corner, except for window 0 in V1-4 where the cursor begins at the bottom left. Initially the screen cursor (at the cursor location of the current window) is on.

Initially the roman style of the standard font is selected for every window (see section 5). The height and width of this style determine the initial value of window property 13.

5. Characters, Fonts, And Styles

Interactive fiction revolves around text, and Z-machine text is divided into characters. This section describes the legal character values, and what the output characters look like.

5.1. Characters

Input and output characters are represented by numbers from 0 to 65535, as indicated by the following table. All numbers not in this table are illegal.

Table 2. Input and output characters

Number	I/O	Meaning
9	O	paragraph indentation (V6 only)
11	O	wide space (V6 only)
13	I/O	newline
32-126	I/O	ASCII characters, or other for a non-ASCII font
127	I	delete
129-132	I	cursor up, down, left, right

133-144	I	function keys F1-12
145-154	I	keypad keys 0-9
155-163	I/O	German special characters
164-251	I/O	unspecified; reserved for other special characters
252	I	in V6 only: menu selection mouse click
253	I	in V6 only: double mouse click
254	I	in V5: any mouse click; in V6: single mouse click
256-767	O	unspecified
768-1023	O	unspecified, but reserved by Graham Nelson
1024-65535	O	unspecified

All numbers marked 'I' or 'I/O' are called 'input characters,' and all marked 'O' or 'I/O' are 'output characters.'

Standard extension: 0 (null) is an output-only, and 27 (escape) is an input-only character.

5.2. Styles and fonts

A style is a set of descriptions which determine what an output character should look like on the screen. A font is a set of related styles. Fonts and their styles are accessed through the I/O card; there is no way for a Z-program to access them directly. It is unspecified what happens when fonts or styles are changed during execution of a Z-program. Recommended behaviour is to reset the screen, and use the new fonts from that time on; resetting the screen is obviously unnecessary when the new characters have the same sizes as the old ones.

A character description occupies a rectangle of units, and says how the character should be shown on the screen. This description refers only to two colours: foreground and background. Note that a unit usually contains either a character or a pixel; see section 4 for details.

A style is a mapping from numbers to character descriptions, covering at least the numbers 32-126 (and in V6 also 9 and 11), and optionally more in the ranges 155-251 and 256-65535. Note that these are the output characters except 13 (newline). In this section we will often use 'character' to mean either the character number or its description.

Note that there are three different ways for the Z-machine to produce output characters, which are described elsewhere in this specification:

- in an alternative character set (8 bits, *i.e.*, 0-255);
- as a literal character in a Z-string (10 bits, *i.e.*, 0-1023);
- in a PRINT_CHAR instruction (16 bits, *i.e.*, 0-65535). [Question: [Nelson] says characters 1024-65535 are not allowed. Is he correct? This seems to be an arbitrary restriction on the `print_char` instruction.]

All characters within a style have the same height, but they need not have the same width. The width of a style is defined to be the width of its character 48 (which is a '0' in an ASCII font; see below). A style is called 'fixed-width' if the width of every character in it is a multiple of the width of its character 48; otherwise it is called 'variable-width.' Note that not all characters in a fixed-width style need have the same width; an example are the two-character equivalents for accented German letters.

Styles are ordered in fonts, and numbered by non-negative integers. All styles in a font consist of the same output characters. If set, the bottom bits of a style number give an indication what the style looks like:

- bit 0: reverse video;
- bit 1: bold;
- bit 2: emphasis;
- bit 3: fixed-width.

All other bits are always %0. Style 0 is called ‘roman.’

A style is called a ‘base style’ for all styles whose numbers can be made by setting zero or more bits to %1 in the base style number. For example: style 5 (reverse video and emphasis) has styles 0 (roman), 1 (reverse video), 4 (emphasis), and itself as its base styles; the roman style is a base style for all styles. Note that the concept of a base style is introduced to cater for Z-machines that do not have all styles available; see the description of the `set_text_style` instruction in section 8.

In V1-3 a font consists of just one style (0, *i.e.* roman), making special effects impossible. In V4+ bits 0, 1, and 3 of a style number may only be %1 if the relevant bits of the header byte at $01 are set.

In V1-4 there is only one font, numbered 1 and called the standard font. If (in V3) the roman style of the standard font is fixed-width, header bit $01/6 should be cleared on start-up; otherwise it should be set. In V5+ there can be more fonts, numbered from 1 upwards:

- 1: standard;
- 2: picture;
- 3: character graphics;
- 4: fixed-width.

The standard font must be present, all others are optional.

5.3. The character descriptions

What should the characters that the I/O card uses look like? The fonts and styles are not specified in detail. Rather, some restrictions are given in this subsection.

The standard and fixed-width fonts are ASCII fonts. Characters 32 to 126 of all styles in an ASCII font faithfully represent the corresponding ASCII characters, as shown in the following table:

	0123456789ABCDEF0123456789ABCDEF
$20	!"#$%&’()*+,-./0123456789:;<=>?
$40	@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
$60	‘abcdefghijklmnopqrstuvwxyz{|}~

Note that character 32 ($20) is a space, and that character 127 ($7F) is not an output character. An ASCII font may also have German characters numbered from 155 to 163. It is legal for these characters to be represented by multi-character ASCII equivalents. Characters 164-252 of an ASCII font are unspecified, but reserved for further specification. These characters might be used for accented characters of languages other than German. All special characters are shown in table 5 in the appendix. Standard extension: characters 164-223 are specified as additional optional special characters; see also table 5 in the appendix.

[Question: I specify above that the multi-character equivalents are just alternative character descriptions. Because character descriptions are always printed as a whole, this implies that these two-character sequences are never split over two lines. Most interpreters that use them, however, will happily split them. Since this case rarely occurs in practice (only with words that don’t fit between the margins), and since I feel that these combinations should never be split, I specify things this way. I think it gives the most ‘natural’ specification, too.]

The picture font is unspecified, because it was never used in the Infocom games; it is recommended not to use it. The character graphics font should look like the (slightly varying between platforms) one supplied with the Infocom game ‘Beyond Zork.’ In the fixed-width font all styles must be fixed-width and have the same width. The fonts numbered from 5 upwards are unspecified. [Question: Font 5 could be reserved for a Unicode or similar ASCII font, which gives the ability to print all kinds of characters in all kinds of languages.]

> *Note:* The behaviour of ’Beyond Zork’ story files w.r.t. character graphics deviates from this specification. First, if the interpreter number (header byte at $1E) is 1, font 3 is always used, ignoring the return value of `set_font`. Second, if the interpreter number is 6 (IBM PC) or 8 (Commodore 64), then ‘Beyond Zork’ assumes that output characters in font 1 are printed using the built-in font of these machines, instead of the Z-machine font. This means that a Z-machine without a character graphics font should never use interpreter numbers 1, 6, and 8. Alternatively, one could use interpreter number 6 or 8, and use a font 1 that resembles the IBM PC or Commodore 64 graphics font, respectively. (This is what ZIP does.)

Within a font, the roman style is the one that is initially used. Bold and emphasis styles are appropriately modified versions of this one; emphasis is usually shown using underlining or italics. A reverse video style is usually the same as one of its base styles, but with the foreground and backgroud colours reversed. A fixed-width style is usually the same as one of its base styles if that is fixed-width. Only if a non-roman style is completely the same as one of its base styles, it is legal (but not mandatory) to use special colours or effects to indicate such characters on the screen.

Characters 9, 11, and 32 are called spaces, and they should look like spaces appropriate for their text style. (This means for instance that character 32 in a reverse video style uses only the foreground colour.) Character 32 has the width of a normal space between words in a sentence.

Character 11 is a wider space, such as commonly used after a period (‘.’) ending a sentence. Character 9 is even wider, and is only used for paragraph indentation. Note that spaces are never split over two lines, since character descriptions are always output as a whole.

[Question: A possible extension is to add a new output character (with a number less than 32?) that looks the same as 32 but is not a space. It can be used to ensure that two consecutive words do appear on the same line.] [Question: Another possible extension is a hyphen that is only printed at the end of a window line, and is otherwise ignored.]

Characters 768-1023 are unspecified in any font, but reserved for further specification by Graham Nelson. These characters might be used as additional special codes for special effects. [Question: Wouldn’t it be better to use unused output characters less than 32 for this? Or aren’t these enough?]

> *Note:* Authors of game-specific emulators are asked to use either output characters 256-767 or 1024-65535, or fonts 5 and higher, for their own special characters.

6. The I/O Card

The I/O card manages all character based input and output of the Z-machine. This includes input from keyboard and mouse, and buffered output. Screen output is sent through the video card; all other input and output is handled by the I/O card itself.

6.1. Output streams

All character based output is performed by sending output character numbers to output streams, of which there are four:

- 1: screen;
- 2: transcript;
- 3: memory;
- 4: command script.

(Output streams 3 and 4 are only available in V3+.) At any given time, an output stream is either open or closed. Anything sent to a closed output stream is simply ignored. There is no way for a Z-program to find out which output streams are open.

Output to stream 1 (and maybe to stream 2, see below) is buffered if and only if the buffer mode of the current window is on.

The following subsections describe for each output stream what happens with the characters sent to them.

6.2. Output stream 1: the screen

Characters sent to stream 1 are to be displayed on the screen, using the video card. To make word-wrap possible, an output buffer is used. All output is first stored in this buffer before it is processed further. The buffer is a sequence of character descriptions with colours; first come spaces, then non-spaces. The width of the buffer is the sum of the widths of its elements. The buffer is initially empty. If buffering is off for the current window, then the buffer is empty. Note that the buffer is added to on outputting a non-newline to output stream 1, and cleared when the buffer is flushed; these are the only ways to change the buffer.

During screen output sometimes an extra newline is generated to avoid crossing the right margin. To record this the I/O card keeps a boolean called ExtraNL meaning ‘the last output character on stream 1 was an extra newline generated by the output process.’ It is used to determine the way to output character 9 in V6 (see below); and to correctly output a sequence of non-spaces wider than a window line. ExtraNL is initially false, and is set to false when the buffer is flushed.

‘Flushing the buffer’ means that the following steps are executed:

```
IF width of buffer > distance from cursor to right margin
AND ExtraNL is false:
  remove all spaces from buffer
  do a NewLine
  set ExtraNL to true
WHILE buffer is not empty:
  remove the first char. description from the buffer and call it c
  WHILE width of c > distance from cursor to right margin:
    do a NewLine
  do a ShowChar(c)
  set ExtraNL to false
```

Note that if buffering is on, the above process in effect replaces all spaces by a newline at the end of a window line. [Question: As far as I can see ZIP and Frotz don’t do this, but instead remove only the first space from the buffer. (Because of this, [Inform] includes a facility that replaces two consecutive spaces by a single one.) Is the above specification the desired behaviour, or are ZIP and Frotz correct?]

When output stream 1 is open and a character is sent to it, it must be shown in the current window. Character 13 (newline) is output by flushing the buffer and performing a NewLine operation. (Standard extension: character 0 (null) is ignored.) All other output characters force a screen change as detailed below.

If in V6 character 9 is to be output, and the cursor is to the right of the left margin, or ExtraNL is true, then character 32 is used instead.

Call *c* the description of the output character in the current font and style, with the current colours. It is unspecified what happens when the character does not occur in that style; recommended behaviour is to use a description representing a question mark (‘?’) with the current style height in the current colours instead. The following steps are now executed:

```
IF c is a space AND buffer contains non-spaces:
  flush the buffer
add c to buffer
IF buffering is off for the current window:
  flush the buffer
```

Non-standard extension used by [Frotz]: It is legal to add the clause “OR *c* is a hyphen” to the condition of the last IF statement above.

> *Note:* Note that some ports of the ITF interpreter (for V5) erroneously uses the buffer mode for window 0 when writing to window 1. Because of these and other bugs, it is recommended that buffering is turned off before writing to window 1 in V3-5.

6.3. Output stream 2: the transcript

Characters sent to stream 2 are sent directly to the outside world. This is intended as a transcript of all output that the Z-program has produced. This specification prescribes nothing about the transcipt; but it is recommended that it ‘looks like’ the output shown on the screen through output stream 1 and picture instructions. [Question: What about sound effects?] Formatting of the transcript may use the buffer mode of the current window.

Note that the Z-machine may sent special messages to stream 2 on certain events, *e.g.*, when displaying a picture on the screen.

Output stream 2 is opened or closed as with the `output_stream` instruction when bit $10-$11/0 is set or cleared.

6.4. Output stream 3: memory

Output characters sent to stream 3 are stored in memory, either in a one-line or multi-line format. The memory address *addr* and the format are given with the `output_stream` instruction that opens stream 3. [**TODO**: Memory contents from *addr* are unspecified while stream 3 is open.]

For the one-line format, the unsigned word at *addr* is the number of characters that have been output, and the bytes after that hold the characters themselves. A new character is stored after the others, and the number of characters is incremented by 1.

For the multi-line format, a number of ‘lines’ are stored from *addr* onwards, followed by a zero word. Every ‘line’ is an unsigned word, followed by that many characters. A new character is stored either after the last of the last ‘line’ (as for the one-line format), or a new ‘line’ is created with the new character as the first. Exception: a character 13 is not stored, and an empty new ‘line’ is created.

The decision whether to go to a new ’line’ depends on information given when the output stream is opened: the text is justified

- as if it was printed to a given window (taking its then-current buffer mode, horizontal cursor position, and margins into account); or
- as if it was printed to a window of a given width (taking the buffer mode of the then-current window into account); this window has the buffer mode of the then-current window, margins coinciding with the window edges, and the cursor at the left window edge.

After opening the stream, changes to the windows’ properties or screen contents have no effect on the way text is justified. [**TODO**: Make this unspecified]

An output character above 255 is either ignored, or replaced by a sequence of byte-valued characters representing it in some way. In V6 the total width of the descriptions of the stored characters in memory is stored in the header word at $30; the descriptions used are the same as would be shown on the screen, in the current font. [Question: Is this also correct for the multi-line format? Or is the maximum width of all lines stored there? Or the width of the last line?] [Question: The details of this above paragraph were invented by me, since [Nelson] is not clear at this point.]

6.5. Output stream 4: the command script

This output stream is intended as a record of all user input to a Z-program. All input is sent through a link to the outside world, and can be retrieved by the Z-machine via input stream 1; see below. This output stream is special in two respects: it handles input characters (as opposed to output characters), and, in V4+, it also handles time-out information. To be precise, all characters and time-out information received in response to `read` and `read_char` instructions are sent to this stream.

> *Note:* Most existing emulators do not write time-out information to a command script.

[Question: There are other factors determining the behaviour of a Z-program except user input: mouse pointer locations, the numbers generated by the random number generator, the interpreter number given by the emulator, finishing of sound effects, the availability of pictures and fonts, etc. Should these be written to a command script too?]

6.6. Input streams

The input streams are

- 0: keyboard;
- 1: command script.

The first is connected to a keyboard, and in V6 an optional mouse; the second reads back information previously written out on output stream 4; see above.

A Z-program has no way of knowing which input stream is used: the Z-machine (probably instructed by the user in some way) may freely switch between the two. The only thing a Z-program can do is to switch to one of these streams using the `input_stream` instruction; but nothing prevents the Z-machine from immediately switching back.

7. The Structure Of Instructions

A Z-code instruction is a sequence of bytes, describing an operation for the Z-machine to perform. Most instructions contain operands. Some instructions are followed by one or more extra arguments. These are not part of the instruction proper, but indicate what should happen with the result of the operation; these arguments are described in detail below. Note that the terms ‘operand’ and ‘argument’ are used consistently to stress the difference.

7.1. The structure of an instruction

Instructions come in different formats, depending on the number of operands they take: short (zero or one), long (two), variable (up to four), double variable (up to eight), and extended (up to four). Double variable instructions are available in V4+, and extended ones in V5+.

Note that the adjective ‘variable’ when applied to instructions has nothing to do with local or global ‘variables’; the former refers to the varying number of operands of an instruction, the latter to a location where a value can be stored.

Note that the ‘extended’ instructions look a lot like variable instructions; this form was created for the V5+ Z-machines, because there were not enough instruction numbers left to squeeze in all the new features.

An instruction consists of two parts: the opcode and the operands. The first one to three bytes contain the opcode and the types of the operands. The following bytes contain the operands themselves. Table 2 in the appendix gives a breakdown of instructions according their ‘operation byte,’ *i.e.*, the byte containing the opcode.

Opcodes consist of a kind and a number; they are denoted as KIND:*n*, where KIND is one of 0OP, 1OP, 2OP, VAR and EXT, and *n* is the opcode number. Instructions of different formats contain different kinds of opcode:

- short: 0OP or 1OP;
- long: 2OP;
- variable: 2OP or VAR;
- double variable: VAR;
- extended: EXT.

Each opcode has a symbolic name – a mnemonic – that describes its effect. Note that some mnemonics in this document differ from those used by [Nelson] and others; table 8 in the appendix summarizes these.

There are three types of operand, which are often indicated by a pair of bits:

- %00: a word constant;
- %01: a byte constant;
- %10: a variable number (a byte).

The absence of an operand is often indicated by %11.

A variable number is a byte that indicates a certain variable. The meaning of a variable number is:

- 0: the top of the routine stack;
- 1-15: the local variable with that number;
- 16-255: the global variable with that number minus 16.

Writing to the variable with number 0 means to push a value onto the routine stack; reading this variable means pulling a value off. If an instruction uses variable number 0 more than once, these operands are processed from left to right. For example,

> `push` ‘constant 5’
> `push` ‘constant 7’
> `sub` ‘variable number 0’ ‘variable number 0’ <*result*>

and

> `sub` ‘constant 7’ ‘constant 5’ <*result*>

are completely equivalent.

At the risk of blurring an issue that might be perfectly clear to the reader, as an example here are three different forms of the `store` instruction. The instruction `store` *var a* says to ‘store the value *a* in the variable with number *var*’ (see 8.2). Thus

> `store` ‘byte constant 3’ ‘byte constant 18’

puts the number 18 in local variable 3 (*i.e.*, the variable with number 3). Instead of putting a constant such as 18 into this variable, we can also, *e.g.*, copy the contents of global variable 2 to it with

> `store` ‘byte constant 3’ ‘variable number 18’ ;

note that number 18 indicates global variable 2. What happens if we

> `store` ‘variable number 3’ ‘byte constant 18’ ?

Using the definition of `store`, this should store the number 18 in the variable with the number that is stored in the variable with number 3. Thus if local variable 3 contains 0, this instruction pushes the number 18 onto the routine stack. If local variable 3 contains 77, this instruction puts the number 18 in global variable 61 (= 77 − 16).

7.2. Decoding an instruction

If (in V5+) the first byte of the instruction is $BE, it is an extended instruction. The second byte contains the EXT opcode. Then follow the operand types (one byte) and the operands themselves, in the same format as for variable instructions (discussed below). It is an error for an extended instruction to occur in V1-4.

Otherwise, the instruction format depends on the top bits of the first byte:

- %0: long;
- %10: short;
- %11: (double) variable.

The first byte of a long instruction is of the form %0*abxxxxx*. Here, %*xxxxx* is the 2OP opcode number, and %*a* and %*b* are abbreviated type-indicators for the two operands:

- %0: a byte constant;
- %1: a variable number.

These correspond to the types %01 and %10, respectively. The next two bytes of the instruction contain the operands, first %*a*, then %*b*.

The first byte of a short instruction is of the form %10*ttxxxx*. Here, %*tt* is the type of the operand (or %11 if absent), and %*xxxx* is the 1OP (or if the operand is absent: 0OP) opcode number. If the operand is present, it follows the first byte.

The first byte of a variable instruction is of the form %11*axxxxx* (except for the two double variable instructions, discussed below), where %*xxxxx* is the (2OP or VAR) opcode number. If %*a* is %0 then this instruction contains a 2OP opcode; if %1, a VAR opcode. It is an error for a variable instruction with a 2OP opcode not to have two operands, except when the opcode is 2OP:$1 (*i.e.*, the instruction is `je`). The second byte is a type byte, and contains the type information for the operands. It is divided into pairs of two bits, each indicating the type of an operand, beginning with the top bits. The following bytes contain the operands in that order. A %11 pair means ‘no operand’; it is an error if a %11 bit pair occurs before a non-%11 pair.

There are two double variable instructions, viz. %11101100 ($EC, opcode VAR:$C) and %11111010 ($FA, opcode VAR:$1A). Their structure is identical to that of variable instructions, except that they do not have a type byte, but instead a type word.

7.3. String, result, and branch arguments

Some instructions are followed by one or more additional arguments. How these arguments are handled is described below. After reading such an argument, the PC is set to the address after it.

Some instructions (viz. `print` and `print_rtrue` ($B2 and $B3)) have a string argument. These instructions are followed by a Z-string.

Some instructions return a result. These instructions are followed by a single byte called a result argument. This byte is the number of the variable where the result should be stored (*i.e.*, it looks like an additional operand of type %10).

Some instructions require a jump (or branch) to be made to another part of the Z-program, depending on the outcome of some test. These instructions are followed by one or two bytes called a branch argument. Bit 7 of the first byte indicates when a branch occurs, a %0 meaning that the branch logic is ‘reversed’: branch if the instruction doesn’t want to, don’t if it does. If bit 6 is %1, the branch argument consists of a single byte and the branch offset is given by its bottom 6 bits (unsigned, *i.e.*, from 0 to 63). If bit 6 is %0, the branch argument consists of two bytes, and the branch offset is given by the bottom 6 bits of the first byte followed by all bits of the second (signed, *i.e.*, from -8192 to 8191).

The following happens when a branch is to be made. If the branch offset is 0 or 1, then instead of branching the instruction `rfalse` or `rtrue`, respectively, is carried out. Otherwise, the branch is made by setting the PC to

$$\text{Address after branch argument} + \text{Branch offset} - 2\ .$$

Note that a branch argument, if present, is always the last of a sequence of arguments.

8. A Catalogue Of Instructions

The last part of this document is perhaps the most difficult: it specifies the effect of all instructions. The difficulty lies in the special cases: what if an instruction occurs only a few times, or never, in existing Infocom game files? And what to do with all kinds of overflow and underflow? Most of the information stated below comes from [Nelson], sometimes supplemented by details found in the source of [ZIP] and [Frotz].

8.1. The entries

Each entry below describes the effect of an opcode. The opcodes are grouped according to their functionality; within each group they are roughly ordered by the version in which they first appear. For reference, table 7 in the appendix lists all opcodes in numerical order.

An instruction is written as a mnemonic, possibly followed by the operands and arguments:

`MNEMONIC` *operands arguments* — *opcode*, *versions*

Here, *opcode* is denoted as KIND:*n*; *versions* is the range of versions for which the instruction is valid, “V1+” being the default. A list of optional operands is surrounded by square brackets (‘[…]’); an initial part of this list must be present. (For example, ‘[*a1 a2 a3*]’ means that either none, or just *a1*, or *a1* and *a2*, or all must be present.) It is an error for an instruction to have more or less operands than stated. The arguments are zero or more of ‘*<string>*’, ‘*<result>*’, and ‘*<branch>*’, always in that order.

The opcode uniquely identifies the instruction format, except in the case of 2OP opcodes. Every 2OP opcode can be used in a long or a variable instruction. However if the instruction is to have a word constant operand, or if there are not exactly two operands, the long format obviously cannot be used. Note that if an instruction with a 2OP opcode can be encoded both ways, the long form is one byte shorter.

All operands are assumed to be unsigned numbers, unless stated otherwise. For some operands, special names are used, sometimes adorned with a suffix. These indicate the kind of operand that is expected there:

- *s*, *t*: a signed number.
- *bit*: either 0 or 1.
- *byte*: byte-valued, *i.e.*, a value between 0 and 255.
- *var*: a variable number, *i.e.*, 0 for the top of the routine stack, 1-15 for local, and 16-255 for global variables (see section 7.1). It is an error if this operand is not byte-valued.
- *baddr*: a byte address.
- *raddr*: a packed address referring to a routine.
- *saddr*: a packed address referring to a string.
- *obj*: a legal object number.

- *attr*: a legal attribute number.
- *prop*: a legal property number, or 0 for the `get_next_prop` instruction.
- *window*: a legal window number, or a special negative value. In the V4+ `erase_window` and the V6 `mouse_window` instructions −1 means the entire screen; in V6 −2 also means the entire screen for an `erase_window`. In V6 −3 is always interpreted as the current window. An optional window operand defaults to the current window.
- *time*: any non-zero number, in tenths of seconds. It is unspecified what happens when this is zero; recommended behaviour is to behave as if no time operand has been given.
- *pic*: a legal picture number. Note that the legality of picture numbers can be checked using the `picture_data` instruction.

It is an error if another value is given for one of the above operands.

Note: The above restrictions are often not reiterated in the instruction descriptions below!

The phrase “The result is…” means that the stated value is stored in the variable given as the result argument. In some descriptions ‘ST’ is used as an operand. This stands for the top-of-routine-stack variable, *i.e.*, the variable with number 0.

8.2. Reading and writing memory

`load` *var <result>* — 1OP:$E

The result is the value of the variable with number *var*.

`store` *var a* — 2OP:$D

Set the variable with number *var* to *a*.

`loadw` *baddr n <result>* — 2OP:$F

The result is the word at $baddr + 2 * n$.

`storew` *baddr n a* — VAR:$1

Store *a* in the word at $baddr + 2 * n$.

`loadb` *baddr n <result>* — 2OP:$10

The result is the byte at $baddr + n$.

`storeb` *baddr n byte* — VAR:$2

Store *byte* in the byte at $baddr+n$.

`push` *a* — VAR:$8

Push *a* on top of the routine stack.

`pull` *var* — VAR:$9, V1-5
`pull` *[baddr] <result>* — VAR:$9, V6

Pull the top off the user stack beginning at *baddr*, and put the result in the variable with number *var* (V6: in the result variable). If no stack is given use the routine stack. In that case it is an error for the routine stack to be empty.

`pop` — 0OP:$9, V1-4

Remove the value on top of the routine stack. It is an error for the stack to be empty.

Note that this opcode is occupied by `catch` in V5+.

`scan_table` *a baddr n [byte] <result> <branch>* — VAR:$17, V4+

Search for the byte or word *a* in a table that begins at *baddr* and is *n* entries long, according to the given format *byte.*

The top bit of the format *byte* is %0 to search for a byte, %1 for a word; the remaining 7 bits give the (unsigned) length of an entry in bytes. The default format is $82, asking to look for a word in a table of words. If it is a table of bytes, it is an error if the first operand is not byte-valued. [Question: An alternative is to always fail the search, and give a warning message.]

The byte (word) *a* is searched for among the first bytes (words) of all entries, in order. If it is found, the result is the first address at which it occurs, and the branch is made; if not, the result is 0, and no branch is made.

It is unspecified what happens when the entry-length is 0, or when it is 1 and a word is searched. Recommended behaviour is to proceed as described above. Since these cases are propably bugs, a warning message should be given if possible.

`copy_table` *baddr1 baddr2 s* — VAR:$1D, V5+

If *s* is positive, copy a region of *s* bytes beginning from *baddr1* to the region of *s* locations beginning at *baddr2*, such that afterwards the second region is equal to the initial state of the first region. (This is important if the regions overlap.) If *s* is negative, $-s$ bytes are copied, and copying proceeds forwards. Exception: if *baddr2* is zero, then the first region is zeroed.

`push_stack` *a baddr <branch>* — EXT:$18, V6

Push *a* onto the user stack at address *baddr* if possible, and branch if successful.

`pop_stack` *n [baddr]* — EXT:$15, V6

Remove *n* words from the top of the user stack beginning at address *baddr*. If no stack is given, use the routine stack. In that case it is an error for the routine stack to contain less than *n* words.

8.3. Arithmetic

`add` *a b <result>* — 2OP:$14

The result is $a + b$ modulo $10000.

`sub` *a b <result>* — 2OP:$15

The result is $a - b$ modulo $10000.

`mul` *a b <result>* — 2OP:$16

The result is $a * b$ modulo $10000.

`div` *s t <result>* — 2OP:$17

The result is floor(s/t) modulo \$10000. It is an error if t is 0. Note that s and t are interpeted as signed numbers.

`mod` *s t <result>* — 2OP:\$18

The result is $s - t * \text{floor}(s/t)$ modulo \$10000. It is an error if t is 0. Note that s and t are interpeted as signed numbers.

`inc` *var* — 1OP:\$5

Increment the value of the variable with number *var* by 1, modulo \$10000. Equivalent to

```
load var ST
add ST 1 ST
store var ST
```

`dec` *var* — 1OP:\$6

Decrement the value of the variable with number var by 1, modulo \$10000. Equivalent to

```
load var ST
sub ST 1 ST
store var ST
```

`inc_jg` *var s <branch>* — 2OP:\$5

Equivalent to

```
inc var
jg var s <branch>
```

`dec_jl` *var s <branch>* — 2OP:\$4

Equivalent to

```
dec var
jl var s <branch>
```

`or` *a b <result>* — 2OP:\$8

The result is the bitwise ‘or’ of *a* and *b*.

`and` *a b <result>* — 2OP:\$9

The result is the bitwise ‘and’ of *a* and *b*.

`not` *a <result>* — 1OP:\$F, V1-4; VAR:\$18, V5+

The result is the bitwise ‘not’ of *a*.

Note that the opcode 1OP:\$F is occupied by `call_p0` in V5+.

`log_shift` *a t <result>* — EXT:\$2, V5+

The result is floor($a * 2^t$) modulo \$10000. Note that *a* is interpeted as an unsigned number.

`art_shift` *s t <result>* — EXT:\$3, V5+

The result is floor($s * 2^t$) modulo \$10000. Note that *s* is interpeted as a signed number.

8.4. Comparisons and jumps

`jz` *a <branch>* — 1OP:$0

Branch if *a* is 0. Equivalent to

`je` *a* 0 *<branch>*

`je` *a [b1 b2 b3] <branch>* — 2OP:$1

Branch if *a* is equal to at least one of the other operands.

Note: Some emulators might not handle this instruction correctly if just one operand is given; correct behaviour is to branch *not*.

`jl` *s t <branch>* — 2OP:$2

Branch if $s < t$. Note that *s* and *t* are interpreted as signed numbers.

`jg` *s t <branch>* — 2OP:$3

Branch if $s > t$. Note that *s* and *t* are interpreted as signed numbers.

`jin` *obj n <branch>* — 2OP:$6

Branch if *n* is the parent object of *obj*, or if *n* is 0 and the object has no parent. Equivalent to

`get_parent` *obj* ST
`je` ST *n <branch>*

`test` *a b <branch>* — 2OP:$7

Branch if the ‘bitwise and’ of *a* and *b* is equal to *b*. Equivalent to

`and` *a b* ST
`je` ST *b <branch>*

`jump` *s* — 1OP:$C

Unconditional branch: set the PC to address after instruction $+ s - 2$. (This strongly resembles the formula for branch arguments; see section 7.3.)

8.5. Call and return, throw and catch

`call_f0` *raddr <result>* — 1OP:$8, V4+
`call_p0` *raddr* — 1OP:$F, V5+
`call_f1` *raddr a1 <result>* — 2OP:$19, V4+
`call_p1` *raddr a1* — 2OP:$1A, V5+
`call_fv` *raddr [a1 a2 a3] <result>* — VAR:$0
`call_pv` *raddr [a1 a2 a3]* — VAR:$19, V5+
`call_fd` *raddr [a1 a2 a3 a4 a5 a6 a7] <result>* — VAR:$C, V4+
`call_pd` *raddr [a1 a2 a3 a4 a5 a6 a7]* — VAR:$1A, V5+

Call the given routine, passing it the given values. Exception: if raddr is 0, either the result is 0 (if there is a result argument) or nothing happens.

Note that the last character of the mnemonic indicates the number of passed values, where `v` (for variable) means 0 to 3, and `d` (for double) means 0 to 7. The character before that indicates whether the instruction has a result argument (`f` for function) or not (`p` for procedure). Note also that V1-3 have only one call instruction, viz. `call_fv`, and that the `call_p` versions exist only in V5+.

The call is executed as follows. Let *r* be the address at which the routine starts (found by 'unpacking' the packed routine address *raddr*); let *L* be the number of local variables of the called routine (found in the byte at address *r*); and let *n* be the number of values that are passed to the called routine (*i.e.*, the number of operands of the instruction minus one). If $n > L$, then read *L* for *n* in the following. (Note that this makes it legal to call a routine with too many values; the extraneous values are simply ignored.) A new frame is pushed on top of the call stack, with the following contents:

– The PC is set to the address of the first instruction of the routine, directly after the routine header (*i.e.*, to $r + 2 * L + 1$ in V1-4, or $r + 1$ in V5+).

– The routine stack is empty.

– *L* local variables are created. The *n* lowest of these are set to the passed values $a1, \ldots, an$. The others are initialized to their default initial values. In V1-4, these are found in the *L* words from address $r + 1$; in V5+, they are 0.

– In V3+, the fact is stored that this routine is called as a function (for the `call_f` instructions) or as a procedure (for the `call_p` variants).

– In V5+, the number *n* is stored.

In this way control is passed to the called routine, until a return instruction is encountered. Note that the PC in the previous frame now contains the address after the call instruction (either pointing to its result argument or to the next instruction).

Note that opcode 1OP:$8 (for `call_f0`) is illegal in V1-3; 1OP:$F is occupied by `not` (instead of `call_p0`) in V1-4. [Question: Why wasn't opcode 1OP:$8 used in V1-3?]

`ret` *a* — 1OP:$B

Return from the current routine with return value *a*. The following steps are executed:

- Check the top frame of the call stack to see whether this routine was called as function, procedure, or interrupt.
- Remove the top frame from the call stack. It is an error for the call stack to be left empty.
- If the routine was called as an interrupt, return the value *a* to the caller.
- Otherwise, if the routine was called as a function, the current PC is pointing to it: store the return value *a* there, moving the PC one byte on to the next instruction. Control has now been passed back to the calling routine.

Note that the following instructions do a `ret` implicitly: `print_rtrue`, `ret_pulled`, `rfalse`, `rtrue`, and `throw`.

`rtrue` — 0OP:$0

Equivalent to `ret` 1.

`rfalse` — 0OP:$1

Equivalent to `ret` 0.

`ret_pulled` — 0OP:$8

Equivalent to `ret` ST.

`check_arg_count` *n* <*branch*> — VAR:$1F, V5+

Branch if the call instruction to the current routine passed at least *n* values. (Remember that the number of values passed to the current routine is stored in the top frame of the call stack.)

Note that it is unspecified what happens when this instruction is encountered in the 'main routine'; recommended behaviour is to branch only if *n* is 0. (Note that ZIP crashes instead.) See also sections 2.2 and 2.11.

`catch` <*result*> — 0OP:$9, V5+

Store a pointer to the frame currently on top of the call stack. Tag the top frame on the call stack with a frame pointer not currently used, and return it as the result (see section 2.2). This frame pointer can then be used by a `throw` instruction.

Note that the `catch` instruction can be 'nested' at most 65536 times, but to reach that maximum a call stack of at least that many frames is needed.

Note that this opcode is occupied by POP in V1-4.

`throw` *a fp* — 2OP:$1C, V5+

Return with return value a from the routine that did the corresponding `catch`; this routine is found using the frame pointer *fp*. *fp* must point to one of the frames currently on the call stack (see section 2.2). The call stack is popped until that frame is on top. Then a `ret` *a* instruction is executed.

> *Note:* `throw`ing the frame pointer of a call stack frame that has been popped since its `catch` leads to nasty bugs. If there currently is no frame with that same frame pointer, the error can theoretically be caught by the emulator (but in practice most do not). If there is one, the error goes unnoticed, leading to strange behaviour later.

8.6. Objects, attributes, and properties

`get_sibling` *obj* <*result*> <*branch*> — 1OP:$1

The result is the (next) sibling object of the given object, or 0 if it doesn't exist. Branch if the result is not 0.

`get_child` *obj* <*result*> <*branch*> — 1OP:$2

The result is the (first) child object of the given object, or 0 if it doesn't exist. Branch if the result is not 0.

`get_parent` *obj* <*result*> — 1OP:$3

The result is the parent object of the given object, or 0 if it doesn't exist. (Note that unlike `get_child` and `get_sibling` this instruction has no branch argument.)

`remove_obj` *obj* — 1OP:$9

Remove *obj* from its current location in the object tree; all its children move with it.

The given object is removed from between its siblings (closing the sibling chain again), and is changed to have no parent and no siblings. If *obj* has no parent – and therefore no siblings – nothing happens.

`insert_obj` *obj1 obj2* — 2OP:$E

Remove *obj*1 from its current location in the object tree, and insert it as the first child of *obj* 2, before all other children. All *obj*1's children move with it.

Object *obj*1 is first removed from its current location, as with `remove_obj` *obj1.* It is then made the (first) child of *obj*2, with the formerly first child as its (next) sibling.

Note that a consistent non-recursive object tree is made recursive by this instruction if and only if *obj*1 occurs in the (finite) parent chain that begins with *obj*2. In this case the emulator might print a warning message, since this is probably a bug.

`test_attr` *obj attr <branch>* — 2OP:$A

Branch if object *obj* has attribute *attr* set.

`set_attr` *obj attr* — 2OP:$B

Set (*i.e.*, make 1, or *true*) attribute *attr* on object *obj*.

`clear_attr` *obj attr* — 2OP:$C

Clear (*i.e.*, make 0, or *false*) attribute *attr* on object *obj*.

`put_prop` *obj prop a* — VAR:$3

Set property *prop* on object *obj* to *a*. The property must be present on the object. If the property length is 1, then *a* must be byte-valued.

`get_prop` *obj prop <result>* — 2OP:$11

The result is the first word (if the property length is 2) or byte (if it is one) of property *prop* on object *obj*, if it is present. Otherwise the result is the default property word stored in the property defaults table. The result is unspecified if the property is present but does not have length 1 or 2.

`get_prop_addr` *obj prop <result>* — 2OP:$12

The result is the address where the property *prop* of object *obj* begins. The property must be present on the object.

`get_next_prop` *obj prop <result>* — 2OP:$13

If *prop* is zero, the result is the number of the first (highest numbered) property on object *obj*. Otherwise, *prop* must be present on this object, and the result is the number of its next (lower numbered) property. In all cases, if no such property is present the result is 0.

`get_prop_len` *baddr <result>* — 1OP:$4

The result is the length (in bytes) of the property starting at address *baddr*. This is stored in some bits of the byte at *baddr* – 1:

- V1-3: the top 3 bits;

- V4+, top bit is %0: bit 6;
- V4+, top bit is %1: the bottom 7 bits.

To compute the length, interpret these bits as an unsigned number, and (except in the third case) add 1.

It is an error if an address is given where no property begins. Note that most emulators (*e.g.*, ZIP) do not check this.

8.7. Windows

`get_wind_prop` *window p <result>* — EXT:$13, V6

The result is the value of property *p* of the given *window*. It is an error if an illegal property number is used.

`put_wind_prop` *window p a* — EXT:$19, V6

Property *p* of the given *window* is set to *a*. It is an error if an illegal property number is used. It is also an error if property number 13 is used. Note that nothing else happens, in particular the output buffer is not flushed. It is recommended to use this instruction only for properties 8, 9, and 15, and use the following instructions for the others: `split_screen` (0-5), `move_window` (0,1), `window_size` (2,3), `set_cursor` (4,5), `set_margins` (6,7), `set_text_style` (10), `set_colour` (11), `set_font` (12), `buffer_mode` (14), `window_style` (14).

`split_screen` *n* — VAR:$A, V3+

Flush the buffer, and position windows 0 and 1 so that they tile the region below the status bar (if present), with window 1 occupying the upper *n* units. (If this is not possible window 1 is made as large as possible, while window 0 gets height 0.) This is done by setting window properties 0 to 3 of windows 0 and 1 accordingly. The cursor positions for windows 0 and 1 (properties 4 and 5) are changed too, such that they remain at the same location on the screen. Exception: If, for windows 0 and 1, the cursor would end up outside the window, the cursor is placed at the top left of the window.

[Question: I think the above is wrong, since *n* is probably given in lines instead of in units. If so: how to convert lines to units – use the height of the current style of window 1?]

It is an error when this instruction is encountered in V3 and header bit $01/5 is cleared.

`set_window` *window* — VAR:$B, V3+

Flush the output buffer and make *window* the currently selected window. When window 1 is selected in V3, it is cleared with an EraseWindow(1) operation; in V4-5 its cursor is set to (1,1). Note that it is legal to select a window with zero size, although it is not possible to print anything to it. It is an error to give a negative value for *window*, except −3 in V6, in which case nothing happens.

`set_cursor` *s x* — VAR:$F, V4-5
`set_cursor` *s x [window]* — VAR:$F, V6

Flush the buffer and set the cursor for the given *window* (*i.e.*, window properties 4 and 5) at position (s, x). It is an error in V4-5 to use this instruction when window 0 is selected. [Question: Is the window operand optional or not?]

Exception: The first argument can be negative. If it is -2 (and the second argument is 0), the screen cursor is turned on. If it is -1, it is turned off; the second argument is ignored in that case.

`get_cursor` *baddr* — VAR:$10, V4+

Flush the buffer, then put the *y* and *x* coordinates of the current window's cursor in the words at addresses *baddr* and *baddr* $+ 2$ respectively. [Question: Maybe 2 should be put in the byte (or word?) at *baddr*, and the *y* and *x* coordinates at the next two words, but PDD says not.] [Question: Graham Nelson suggests as an alternative to let the result of a `get_cursor` be unspecified if the buffer is not empty. I think the above is a cleaner solution.]

`buffer_mode` *bit* — VAR:$12, V4+

Flush the output buffer and set the buffer mode (attribute 4) of both windows (V4), of window 0 (V5), or of the current window (V6) to *bit*.

`set_colour` *byte0 byte1* — 2OP:$1B, V5+

Set the colour property (11) to $256 * byte0 + byte1$ for both windows (V5) or the current window (V6); the meaning of the colour values is given in section 4.3. It is an error if another colour value is used. Exceptions: Three special colour values can be given, viz.

- 0: Don't change the colour, *i.e.*, use the current colour;
- 1: Set to the default colour in the header byte at $2C or $2D;
- 255: Set to the foreground colour of the unit at the current window's cursor position.

[Question: Should the output buffer be flushed before changing colours? I think not.]

`set_text_style` *n* — VAR:$11, V4+

Set the text style (property 10) for all windows (V4-5) or for the current window (V6), to 0 if *n* is 0, and to current_text_style 'bitwise or' *n* otherwise.

Note that also property 13 and the header words at $26 and $27 must be changed; these store the size of the current style.

Note that the output buffer is not flushed.

`set_font` *n <result>* — EXT:$4, V5
`set_font` *n [window] <result>* — EXT:$4, V6

Set the font property (12) to *n*, if it is available; otherwise do nothing. In V5 the font is set for all windows; in V6 it is set for the given *window* only. The return value is the previous value of the font property, or 0 if the requested font was not available. [Question: Is the window operand optional or not?] [Question: Is a `set_text_style` 0 also performed?]

Note that also property 13 and the header words at $26 and $27 must be changed; these store the size of the current style.

Note that the output buffer is not flushed.

`move_window` *window y x* — EXT:$10, V6

Flush the output buffer and set the coordinates of the top left of the given *window* (*i.e.*, properties 0 and 1) to *y* and *x*, respectively. [Question: What happens with the cursor position? It should remain at the same relative or absolute position. In the latter case, if the cursor would end up outside the window, the cursor should be placed at the top left corner (cf. `split_screen`).]

`window_size` *window y x* — EXT:$11, V6

Flush the output buffer and set the vertical and horizontal size of the given *window* (*i.e.*, properties 2 and 3) to *y* and *x* respectively. [Question: What happens if the cursor is outside the resized window? I suggest setting it to the top left corner.]

`set_margins` *xl xr [window]* — EXT:$8, V6

Flush the output buffer and set the margin properties (6 and 7) of the given *window* to *xl* and *xr* respectively. [Question: And set the x coordinate of the cursor (property 1) to *xl*? Graham Nelson says not.] [Question: Is the window operand optional or not?]

`window_style` *window flags op* — EXT:$12, V6

Set the attribute property (14) of the given *window* to a new value, depending on *op*:

- 0: *flags*;
- 1: the 'bitwise or' of *current* and *flags*;
- 2: the 'bitwise and' of *current* with the 'bitwise not' of *flags*;
- 3: the 'bitwise exclusive or' of *current* and *flags*.

8.8. Input and output streams

`output_stream` *s* — VAR:$13, V3-4
`output_stream` *s [baddr]* — VAR:$13, V5
`output_stream` *s [baddr w]* — VAR:$13, V6

If $s > 0$ then open output stream *s*; if $s < 0$ then close output stream $-s$. It is an error for *s* to be zero, or an illegal output stream number. [Question: [Nelson] declares this to be legal. Why?]

Only if *s* is 3, the *baddr* operand must be given. This is the address where storage begins. If the optional *w* operand is not present, the one-line format is used; if it is, the multi-line format. In the last case, if $w \geq 0$ then the properties of window *w* is used for text justification; if $w < 0$ then an imaginary window with width $-w$ is used. See section 6.4 for further details.

`input_stream` *n* — VAR:$14, V3+

Switch to input stream *n*. It is an error to give an illegal input stream number.

8.9. Input

`read` *baddr1 baddr2* — VAR:$4, V1-3
`read` *baddr1 baddr2 [time raddr]* — VAR:$4, V4
`read` *baddr1 baddr2 [time raddr] <result>* — VAR:$4, V5+

First refresh the status bar. Then read a sequence of characters from the current input stream, simultaneously displaying it on the screen, and optionally calling a 'time-out routine' at intervals. Finally tokenise it using the main dictionary.

First, if a status bar is present, a `show_status` instruction is performed. The now-current state of the Z-machine and the screen (call it *S*) is remembered. Note that this remembered state plays a crucial role in the explanation below.

A sequence of input characters is now constructed by reading the device associated with the current input stream (*i.e.*, the keyboard or a command script). How this construction is done is not specified, but it usually begins with the empty sequence, and it is legal to edit the input sequence in any way imaginable (using, *e.g.*, delete, insert, and cursor keys). It is also legal to limit the input line to a certain length, *e.g.*, to the length of the current window line, or to the length of the input buffer (see below).

During the construction process, but only if output stream 1 is open, it is mandatory show the sequence constructed thus far on the screen. To be precise, at any given time the screen should look as if the current sequence was printed using `print_char` instructions from state *S*. (Note that the `print_char` instructions are not actually performed, so that only the screen changes. In particular, nothing is sent to any output stream.)

Construction ends when a so-called 'terminating character' is entered by the user. Character 13 (newline) is always a terminating character. In V5+ the header word at $2E contains either zero, or the byte address of a zero-terminated list of additional terminating characters; an entry of 255 in this table means that any input-only character (*i.e.*, any 'function key') terminates input. (Input-only characters that are not terminating and have no special meaning to the interpreter are simply ignored.) [Question: If the delete key is a terminating character, and it is pressed during construction, one can use it (i) to edit the sequence under construction (ii) as a terminating character. Option (iii) is to make it illegal for 127 to be a terminating character. I prefer (ii).]

In V4+ it is possible to interrupt the construction process if it takes longer than a given time. To achieve this, the *time* and *raddr* operands must both be given. (It is an error if only the *time* operand is given.) In this case the following happens after *time*/10 seconds have elapsed. First the screen is reverted to state *S*. Then the routine at the given packed address is called as an interrupt, without passing it any values. If the return value is zero, the remembered state *S* is replaced by the then-current Z-machine and screen state, and construction continues (with the input sequence being shown as before). If the return value of the time-out routine is non-zero, the constructed sequence is made empty, and the construction process is ended with 0 as the terminating character. [Question: Is it an error for the time-out routine to return another value than 0 or 1?]

Note: Some versions of ZIP erroneously call the routine every time seconds, and pass time to it.

After the construction process has terminated, the screen is reverted to state *S*, and the sequence of input characters is sent to all output streams. If the terminating character is 13, it is also sent to all output streams.

The sequence of input characters (excluding the terminating character) is stored in the buffer that begins at address *baddr* 1, but with upper case letters converted to lower case. Call the unsigned byte at that address *n*. In V1-4, the characters are stored in the bytes from address *baddr*1+1 upwards, and followed by a zero byte. In V5+, the buffer already contains a sequence of characters, beginning at address *baddr*1 + 2, with the number of characters stored as an unsigned byte at *baddr*1+1. The sequence just read is stored after the characters already present, and the total number of characters is stored in the byte at *baddr*1 + 1. The maximum number of characters that may be stored in the buffer is *n* – 1 (V1-4) or *n* (V5+). It is an error to attempt to store more characters. (Note that in practice, the construction process should be such that this limit cannot be exceeded.)

Finally, if *baddr*2 is not zero in V5+, the sequence stored in the buffer is tokenised, just as if a `tokenise` *baddr1 baddr2* instruction was used.

In V5+ the result is the terminating character.

Note that it is legal to execute a `read` instruction in any window.

`read_char` *1 [time raddr] <result>* — VAR:$16, V4+

One input character is read from the current input stream, and is returned as the result. In V4+ the *time* and *raddr* operands may (both) be given, just as with the `read` instruction. They are used in the same way: on a return value of 0 reading continues; otherwise the instruction finishes without reading a character, and the result is 0. [Question: Should the read character (and possible time-out information) be output to output stream 4?]

Note that the first operand must be 1. The meaning of this operand is not known.

8.10. Character based output

`print_char` *n* — VAR:$5

Output character *n* to all output streams except 4. [Question: [Nelson] says characters 1024-65535 are not allowed. Is he correct? This seems to be an arbitrary restriction on the `print_char` instruction.]

`new_line` — 0OP:$B

Equivalent to `print_char` 13.

`print` *<string>* — 0OP:$2

Convert the Z-string given as the string argument to a sequence of output characters (see section 3.2), and output this sequence as if `print_char` instructions were used.

`print_rtrue` *<string>* — 0OP:$3

Equivalent to

```
print <string>
new_line
rtrue
```

`print_addr` *baddr* — 1OP:$7

Like `print`, but using the Z-string beginning at address *baddr*.

`print_paddr` *saddr* — 1OP:$D

Like `print`, but using the Z-string beginning at the packed string address *saddr*.

`print_num` *s* — VAR:$6

Convert *s* to a list of output characters representing a signed decimal number, and output these characters in sequence as with the PRINT_CHAR instruction. No leading or trailing spaces are printed, and a sign is only printed if the number is negative.

`print_obj` *obj* — 1OP:$A

Like `print`, but using the name of object *obj* stored as a Z-string in the header of its property list.

`print_table` *baddr x [y n]* — VAR:$1E, V5+

Print a rectangle of output characters stored from address baddr on the screen, with width *x* and height *y* (default 1). The characters are printed using `print_char`, and inbetween lines the cursor is moved to a new line using `set_cursor` with appropriate arguments. [Question: Should the cursor end up after the last character of the last line, or on the left hand side of the rectangle on the line below it? Frotz does the former, but the latter might be more appropriate, at least for $y > 1$. It might be best to leave this unspecified.] [Question: Is it an error for *y* to be 0, or should nothing be done in that case? I prefer the latter.]

The characters to be printed are stored in the bytes beginning at *baddr*. First come *x* characters to be printed, then *n* (default 0) characters to be ignored, again *x* to be printed, *etc*.

`print_form` *baddr* — EXT:$1A, V6

Print a number of ‘lines’ stored in memory, in the same way as stored using output stream 3 (see section 6.4).

The ‘lines’ begin at address *baddr*, and are followed by a zero word. Every ‘line’ is an 2-1-table of output characters. For every line, these characters are output using `print_char`, and at the end of each line a `new_line` is output.

`scroll_window` *window s* — EXT:$14, V6

Execute the ScrollWindow(*window*,*s*) operation. [Question: This moves the cursor also. Is that correct?]

8.11. Miscellaneous screen output

`erase_line` — VAR:$E, V4-5
`erase_line` *[n]* — VAR:$E, V6

Erase a rectangle of the current window with the cursor position as its top left corner; use the current background colour. [Question: Is this correct for all versions?] The height of this rectangle (in units) is the height of the current style in V4-5, and 1 in V6. Its width is $n - 1$ if *n* is greater than 1; otherwise it is the distance from the cursor position to the right margin. *n* defaults to 1. [Question: Is the operand mandatory in V6? Is 0 an illegal value for it?]

`erase_window` *window* — VAR:$D, V4+

If *window* is not negative, perform an EraseWindow(*window*) operation. If -1: perform an EraseScreen operation followed by a `split_screen` 0 instruction. If -2 in V6: perform an EraseScreen operation.

`draw_picture` *pic [y x]* — EXT:$5, V6

Do a ShowPicture(*pic*,*y*,*x*) operation on the video card. If *y* and *x* are not given, they default to the cursor position of the current window. It is an error if only *y* is given.

`erase_picture` *pic [y x]* — EXT:$7, V6

Do an ErasePicture(*pic*,*y*,*x*) operation on the video card. If *y* and *x* are not given, they default to the cursor position of the current window. It is an error if only *y* is given.

`picture_data` *n baddr <branch>* — EXT:$6, V6

If *n* is a legal picture number, store the height and width of that picture in two words at address *baddr* and branch. If *n* is zero, store the highest legal picture number in the word at *baddr*, and the release number of the picture file in the next word. It is an error if *n* has any other value. [Question: Should these be 2-2-tables? PDD says not.]

`picture_table` *baddr* — EXT:$1C, V6

Do nothing. This hints the Z-machine that some pictures will be shown soon, using a 2-2-table at address *baddr* which contains the picture numbers as unsigned numbers.

8.12. Sound, mouse, and menus

`sound` *n [op vol raddr]* — VAR:$15, V3+

This instruction has a complicated semantics, which depends on the sound number *n*. If this is 1 or 2, a high-pitched or low-pitched beep is played, independently of running sound effects; it is an error to give more operands. If $n > 2$ the *op* operand must be present, and lie between 1 and 4. This gives the action to be taken:

- 1, prepare: the sound effect will soon be started.
- 2, start: if the required sound effect is already running, change its repeats, volume, and interrupt routine; otherwise stop the current sound effect, and (if it can be played) start the required one, with the given repeats, volume, and interrupt routine. The high byte of the *vol* operand is the number of repeats (default 0, always 0 in V3), the low byte is the volume (must be 1-8 or 255, default 255). [Question: Is the default volume correct?] ‘0 repeats’ really means ‘play once’; ‘255 repeats’ means ‘repeat indefinitely’; and ‘volume 255’ means ‘the loudest possible.’ Note that the interrupt routine is only called when the sound effect has been played completely the required number of times. The routine is called without passing it any values, ignoring the return value.
- 3, stop: stop the current sound effect. The sound number is ignored.
- 4, finish with: the sound effect will not be used for some time.

It is legal for *n* to be zero. The *vol* and *raddr* operands are only allowed when $n > 2$ and *op* $= 2$.

It is legal to postpone starting a new sound effect until the current one has ended, if no input has been read from any input stream since the current sound effect has been started. The interrupt routine for the current sound effect is not called in this case. (This is to cater for “The Lurking Horror”s assumption about the low speed of its interpreter.)

Note that “The Lurking Horror” sometimes uses this instruction in a complicated and presumably buggy way.

`read_mouse` *baddr* — EXT:$16, V6

Read the current state of the mouse, and write them in four words at *baddr*. The words contain the *y* and *x* coordinates, a button word, and a menu word. The button word contains a %1 bit for every mouse button pressed, with bit 0 representing the rightmost button, bit 1 the next, etc. The high byte of the menu word is the menu number, the low byte is the item number. [Question: Since the mouse state includes a menu choice, I presume that this state is refreshed only at mouse clicks. Is that correct?]

`mouse_window` *window* — EXT:$17, V6

Restrict the mouse pointer to the given window, or remove the current restriction if *window* is −1; see section 2.10.

`make_menu` *n baddr <branch>* — EXT:$1B, V6

Create or replace the menu with number *n*, or remove that menu if *baddr* is zero.

The menu is stored as a sequence of ASCII strings. The first string is the menu title, the others are the menu entries. The word (byte?) at *baddr* stores the number of strings, and is followed by that many strings. Each string is an unsigned word (byte?) followed by that many ASCII character bytes.

8.13. Save, restore, and undo

`save` *<branch>* — 0OP:$5, V1-3
`save` *<result>* — 0OP:$5, V4
`save` *[baddr1 n baddr2] <result>* — EXT:$0, V5+

Save the dynamic memory plus the call stack, or the part of memory beginning at *baddr*1 of *n* bytes long, via a link to the outside world. (It is an error if only the *baddr*1 argument is present.) The optional 1-1-table of ASCII character bytes at *baddr*2 gives a suggested name for the location of the saved data.

In V1-3, if the save operation failed or on a succesful restore, execution continues after the `save` instruction; if the save operation succeeded the branch is made. In V4+ the result is 0 for failure, 1 for success, and 2 for succesful restore, and execution always continues after the `save` instruction.

If the call stack is saved, it is illegal for it to contain interrupt frames. This implies that it is illegal to use this instruction from within a routine called as an interrupt.

It is legal for the Z-machine to have a dialogue with the user about the location of the saved data, and show it in window 0. [Question: Or should that be the current window?]

`restore` *<branch>* — 0OP:$6, V1-3
`restore` *<result>* — 0OP:$6, V4
`restore` *[baddr1 n baddr2] <result>* — EXT:$1, V5+

Restore the dynamic memory plus the call stack, or the part of memory beginning at *baddr*1 of *n* bytes long, via a link from the outside world. (It is an error if only the *baddr*1 argument is present.) The optional 1-1-table of ASCII character bytes at *baddr*2 gives a suggested name for the location of the restored data.

Just as when the Z-machine is initialized, header bit \$10-\$11/0 is not changed.

If the restore operation succeeds and the call stack is restored (*i.e.*, no operands are given), execution continues after the corresponding `save` instruction, in the restored environment (and its result is 2). Otherwise, execution continues after this `restore` instruction; the V4+ result is the number of bytes restored, or 0 if the restore failed.

Note that the V1-3 branch argument is ignored.

If the call stack is restored, it may not contain interrupt frames (see `save` above).

It is legal for the Z-machine to have a dialogue with the user about the location of the saved data, and show it in window 0. [Question: Or should that be the current window?]

`save_undo` <*result*> — EXT:\$9, V5+

Like V4 `save`, but saving to the save memory instead of to the outside world. The result is −1 if the Z-machine has no save memory. [**TODO**: Add something on multiple undo?]

Note that in V5+ opcode 0OP:\$5 is illegal.

`restore_undo` <*result*> — EXT:\$A, V5+

Like V4 `restore`, but restoring from the save memory instead of from the outside world. (The result is 0 if the save memory is not initialized, *i.e.*, if there was no previous `save_undo`.) The result is −1 if the Z-machine has no save memory.

Note that in V5+ opcode 0OP:\$6 is illegal.

8.14. Miscellaneous

`nop` — 0OP:\$4

No OPeration, *i.e.*, do nothing.

`random` *s* <*result*> — VAR:\$7

If $s > 0$, the result is a random number between 1 and *s*, inclusive. If $s < 0$, the random generator is seeded with *s* and the result is 0. If *s* is 0, all future random numbers will be 'really random,' and the result is 0. See also section 2.6.

Note: Early ports of ZIP do not reseed the random generator when *s* is 0.

`restart` — 0OP:\$7

Initialize the Z-machine, thereby restarting the original Z-program; see section 2.11 for a description of what exactly happens.

`quit` — 0OP:\$A

The Z-machine is halted.

`show_status` — 0OP:\$C, V3

Refresh the status bar by performing a ShowStatusBar(*s*,*a*,*b*,*flag*) operation. Global variable 0 must contain a legal object number; the name of that object (*i.e.*, the Z-string in the header of its property list) is converted to a sequence of output characters and called *s*; it is an error if *s* contains a newline. *a* and *b* are the (signed) values stored in globals 1 and 2 respectively. In V3 *flag* is header bit \$01/1; otherwise it is %0.

Note that this instruction is implicitly used in the V1-3 READ instruction.

`verify` *<branch>* — 0OP:$D, V3+

Compute a checksum from the non-header bytes, and branch if this matches the checksum in the header.

Let *n* be the number stored in the header word at $1A, and *f* be the ‘program length scale factor’: 2 for V3, 4 for V4-5, and 8 for V6-8. This instruction sums (modulo $10000) the bytes from addresses $40 upwards to $n * f - 1$ inclusive. This sum is compared with the header word at $1C, branching if equal. The sum is defined to be zero if $n * f - 1$ is less than $40; this means that the branch is always made if the header words at $1A and $1C are both zero, which is the case in early V3 Infocom games.

`piracy` *<branch>* — 0OP:$F, V5+

This is a conditional branch instruction, but the condition is unspecified. Originally this was supposed to branch if the emulator thinks an official copy of the current Z-program is used. Because most emulators probably have no means to check this, it is recommended that this instruction should always branch.

`tokenise` *baddr1 baddr2 [baddr3 bit]* — VAR:$1B, V5+

Tokenise the input characters stored in the text buffer at $baddr1$, into the parse buffer at $baddr2$. If $baddr3$ is present and non-zero, use the dictionary at that address; otherwise use the main dictionary.

The input characters are stored in the bytes from address $baddr1 + 1$ with a zero terminator (V1-4), or from address $baddr1 + 2$ with the number of bytes in the byte at $baddr + 1$ (V5+). Note that this is the format that is produced by `read`.

The text is divided into tokens. Division takes place at characters 32 (spaces) and at the separator characters stored in the header of the dictionary; the former are ignored, the latter are made into separate tokens. [Question: What happens if 32 is used as a separator? I suggest it is made into a token just like any other separator.] As an example, if the comma “,” is a separator character, the text fred,go fishing is divided into the four tokens “fred”, “,”, “go”, and “fishing”. For each token the starting location *p* (w.r.t. $baddr1$) and the number of characters *n* is remembered. Each token is converted into a Z-string, as with the `encode_text` *baddr1 n p* instruction. This Z-string is looked up in the dictionary, and the address of its entry is remembered.

The first byte of the parse buffer gives the maximum number of tokens; it is an error if the text buffer contains more tokens. The actual number of tokens is stored in the next byte, and subsequently a 4-byte block follows for every token. A block consists of the byte address of the dictionary entry (or zero if the token doesn’t occur in it); the length *n* of the token (a byte); and the location *p* of the token (also a byte). Exception: If the *bit* operand is present and non-zero, 4 bytes are skipped for every token not in the dictionary (instead of storing a 4-byte block). Note that this can be used to incrementally tokenise a text using different dictionaries having the same separators.

Note that this instruction is used implicitly to do the lexical analysis for `read` in all versions. That is the reason for mentioning V1-4 in the above description.

`encode_text` *baddr1 n p baddr2* — VAR:$1C, V5+

Create a Z-string that converts to as long an initial part of the given sequence of output characters as possible. This sequence is *n* bytes long, and begins at address $baddr1 + p$; it may contain all output characters less than 256, except 13 (newline). The resulting Z-string is stored from address *baddr*2; it must have the same structure as one in a dictionary: it is 4 (V1-3) or 6 (V4+) words long and padded out with Z- character 5. The encoding should use the built-in or alternative character set as much as possible; it should not use abbreviations or multiple shift Z-characters (V1-2: 2,3; V3+: 4,5). [Question: Did Infocom interpreters really not use the latter? If they did, change this into a 'strong recommendation.'] Note that there may be different Z-strings that encode the given output characters; in such a case it is unspecified which is chosen. [Question: Or is this an error?]

Note that this instruction is used implicitly by `tokenise`, which is implicitly used by `read` in all versions. That is the reason for mentioning V1-4 in the above description.

Miscellaneous Tables

Below, some tables are collected for easy reference. Most of the information in these tables is copied from [Nelson].

1. The header

Table 3 gives the meaning of all bits in the 64-byte header. All addresses are given as hexadecimal numbers. The 'Ver' column gives the range of versions in which the header entry is used, with 1+ as the default. The 'Mem' column gives the type of the locations, the default being ROM.

Table 3. The header

Addr	Ver	Mem	Description
00			version number
01/0	—	—	[unused, maybe intended for reverse byte order]
01/1	3		status line type (0=score/turns, 1=hours:minutes)
01/2	3		[unknown, always set in V3]
01/3	3	IROM	run program in censored mode?
01/4	3	IROM	status line not available?
01/5	3	IROM	upper window available?
01/6	3	IROM	is the default font variable-width?
01/7	—	—	[unused by Infocom, used in Standard]
01/0	5+	IROM	colours available?
01/1	4	IROM	inverse video styles available?
	5	IROM	picture displaying available?
01/2	4+	IROM	boldface styles available?
01/3	4+	IROM	emphasis styles available?
01/4	4+	IROM	fixed-width styles available?
01/5	6	IROM	sound effects available?
01/6	—	—	[unused]
01/7	—	—	[unused by Infocom, used in Standard]
02-03			release number

04-05			begin of paged memory (byte address)
06-07	1-5		first instruction (byte address)
	6		first routine (packed routine address)
08-09			begin of dictionary (byte address)
0A-0B			begin of object table (byte address)
0C-0D			begin of global variables table (byte address)
0E-0F			begin of static memory (byte address)
10-11/0		RAM	set to turn transcripting on
10-11/1	3-4	RAM	set to force printing in fixed-width style
10-11/2	6	RAM	must status line be redrawn?
10-11/3	5+	IROM	does program want to use pictures?
10-11/4	5+	IROM	does program want to use the UNDO instruction?
10-11/5	5+	IROM	does program want to use a mouse?
10-11/6	5+	IROM	does program want to use colours?
10-11/7	5+	IROM	does program want to use sound effects?
10-11/8	5+	IROM	does program want to use menus?
10-11/9	—	—	[unused]
10-11/A	?	IROM	[maybe set on a transcription error; not in V4+]
10-11/B	—	—	[unused]
10-11/C	—	—	[unused]
10-11/D	—	—	[unused]
10-11/E	—	—	[unused]
10-11/F	—	—	[unused]
12-17	2+		serial code (ASCII, usually a date YYMMDD)
18-19	2+		begin of abbreviations table (byte address)
1A-1B	3+		file length (divided by 2 (V1-3), 4 (V4-5), or 8 (V6-8))
1C-1D	3+		file checksum
1E	4+	IROM	interpreter number (see below)
1F	4+	IROM	interpreter version (V4-5: ASCII; V6: number)
20	4+	IROM	screen height (lines)
21	4+	IROM	screen width (characters)
22-23	5+	IROM	screen width (units)
24-25	5+	IROM	screen height (units)
26	5+	IROM	height of current font&style (units)
27	5+	IROM	width of current font&style (units)
28-29	6		routines offset (divided by 8)
2A-2B	6		string offset (divided by 8)
2C	5+	IROM	default background colour number (2-9)
2D	5+	IROM	default foreground colour number (2-9)
2E-2F	5+		begin of terminating char's table (byte address)
30-31	6	IROM	width of text sent to output stream 3 (units)
32-33	—	—	[unused by Infocom, used in Standard]
34-35	5+		begin of alternative char. set (byte address)
36-37	5+		begin of extension table (byte address)
38-3F	6	IROM	login name of player (ASCII)

The rest of this section describes the allowed values for the header entries. All bits and bytes that are unused by a Z-program (*e.g.*, because the entry is not valid for its version, or because the program doesn't need certain tables) must be 0.

At start-up or restart the Z-machine sets all header entries for its version marked 'IROM' (except the word at $30) to initial values depending on the configuration.

Bit $10-$11/0 is initialised to %0. The Z-program sets and clears this to open and close output stream 2. This is also changed on an `output_stream` 2 or `output_stream` −2 instruction.

[Question: What does bit $10-$11/1 mean exactly?]

Bit $10-$11/2 is initialised to %0, and set by the Z-machine when it thinks the status line on the screen needs to be redrawn. The Z-program is expected to detect this, do the redraw, and clear the bit.

Bits $10-$11/3-8 should be set in a Z-program if the program wants to use these features. On initialisation the Z-machine should clear the bits for features it cannot provide. [Question: [Nelson] makes an exception for bit 6, which indicates colours. Why?]

It is not known what bit $10-$11/$A was used for. It is recommended that this bit is set to zero at initialisation, and left alone.

The file length given in entry $1A-$1B need not be exactly the same as the length of the Z-program, but it may also be less (even zero). In particular, the length of a Z-program is not limited by the largest file length that can be stored here. The file checksum should be such that the `verify` instruction succeeds.

The interpreter numbers ($1E) used by Infocom are listed in the following table.

Table 4. Interpreter numbers

No.	Description
1	DECSystem-20[1]
2	Apple][e
3	Macintosh
4	Amiga
5	Atari ST
6	IBM PC
7	Commodore 128
8	Commodore 64
9	Apple][c
10	Apple][gs
11	Tandy Color

An interpreter should choose a number most suitable for the machine it runs on; this might imply choosing a new number above 11. Most ports of the ITF interpreter use 2; most ports of ZIP

[1]This was Infocom's own mainframe.

use 6. (Note that the behaviour of ‘Beyond Zork’ w.r.t. character graphics in fact depends on the interpreter number; see section 5.3 for more information.)

[Question: To what values are $20 and $21 to be set, if a unit represents, *e.g.*, a pixel?]

The width and height of the current font and style are set to that of the initial font and style of the initial window, and kept up-to-date.

The default colours ($2C-$2D) must only be set when the Z-machine can handle colours.

The user name ($38-$3F) is best ignored. Note that $3C-$3F is interpreted differently in the Standard extension described below.

Standard extension: As a non-Infocom extension, Graham Nelson proposes to use the following hitherto unused header entries:

Table 5. The Standard header extensions

Addr	Ver	Mem	Description
01/7	4+	IROM	input time-out available?
32-33		IROM	interpreter revision number (unsigned, unsigned)
3C-3F		IROM	Inform compiler version (ASCII, *e.g.*, “6.01”)

[Question: It would be more consistent to make $01/7 mean ‘input time-out *not* available,’ wouldn’t it? Also, I think things such as the Inform compiler version are better placed in the ‘extension table’ (see 3.9).]

The Z-machine writes its revision number into $32-$33 on initialisation. (Note that this makes the revision number of all Infocom interpreters 0.0.) This can be used by a Z-program to check which features it can expect.

2. The instructions by operation byte

This table gives a breakdown of instructions w.r.t. their ‘operation byte,’ *i.e.*, the byte that contains the opcode number. This is either the first byte, or the second in case of an extended instruction. Note that the opcode number can be computed by subtracting the first number in the operation byte range from the operation byte.

Table 6. The instructions by operation byte

Operation byte	Opcode	Remarks
$00-$1F	2OP:$0-$1F	Both operands are byte constants
$20-$3F	2OP:$0-$1F	Operands are byte constant, variable number
$40-$5F	2OP:$0-$1F	Operands are variable number, byte constant
$60-$7F	2OP:$0-$1F	Both operands are variable numbers
$80-$8F	1OP:$0-$F	Operand is word constant
$90-$9F	1OP:$0-$F	Operand is byte constant

$A0-$AF	1OP:$0-$F	Operand is variable number
$B0-$BF except $BE	0OP:$0-$F	$BE indicates extended instruction
$C0-$DF	2OP:$0-$1F	Variable instruction format
$E0-$FF	VAR:$0-$1F	
$00-$FF after $BE	EXT:$0-$FF	

3. The instructions by opcode

The following table gives the instruction corresponding with each opcode.

The version column says for which versions the opcode is valid; the default value is 1+. If different from the lowest of these, a number between parentheses gives the first version in which this opcode is actually used; a “(–)” means that it is not found in any existing Infocom story file. A “—” means that this opcode is not used in any version.

The last column gives the mnemonical notation of the instruction and its arguments, identical to that used in section 8.

Table 7. Instructions by opcode

Opcode	Version	Mnemonic, operands and arguments
0OP:$0		`rtrue`
0OP:$1		`rfalse`
0OP:$2		`print` <*string*>
0OP:$3		`print_rtrue` <*string*>
0OP:$4	(–)	`nop`
0OP:$5	1-3	`save` <*branch*>
	4	`save` <*result*>
0OP:$6	1-3	`restore` <*branch*>
	4	`restore` <*result*>
0OP:$7		`restart`
0OP:$8		`ret_pulled`
0OP:$9	1-4	`pop`
	5+ (–)	`catch` <*result*>
0OP:$A		`quit`
0OP:$B		`new_line`
0OP:$C	3	`show_status`
0OP:$D	3+	`verify` <*branch*>
0OP:$E	—	[the first byte of an extended instruction in V5+]
0OP:$F	5+ (–)	`piracy` <*branch*>
1OP:$0		`jz` *a* <*branch*>
1OP:$1		`get_sibling` *obj* <*result*> <*branch*>
1OP:$2		`get_child` *obj* <*result*> <*branch*>
1OP:$3		`get_parent` *obj* <*result*>
1OP:$4		`get_prop_len` *baddr* <*result*>

1OP:$5		`inc` *var*
1OP:$6		`dec` *var*
1OP:$7		`print_addr` *baddr*
1OP:$8	4	`call_f0` *raddr <result>*
1OP:$9		`remove_obj` *obj*
1OP:$A		`print_obj` *obj*
1OP:$B		`ret` *a*
1OP:$C		`jump` *s*
1OP:$D		`print_paddr` *saddr*
1OP:$E		`load` *var <result>*
1OP:$F	1-4 (4)	`not` *a <result>*
	5+	`call_p0` *raddr*
2OP:$0	—	
2OP:$1		`je` *a [b1 b2 b3] <branch>*
2OP:$2		`jl` *s t <branch>*
2OP:$3		`jg` *s t <branch>*
2OP:$4		`dec_jl` *var s <branch>*
2OP:$5		`inc_jg` *var t <branch>*
2OP:$6		`jin` *obj n <branch>*
2OP:$7		`test` *a b <branch>*
2OP:$8		`or` *a b <result>*
2OP:$9		`and` *a b <result>*
2OP:$A		`test_attr` *obj attr <branch>*
2OP:$B		`set_attr` *obj attr*
2OP:$C		`clear_attr` *obj attr*
2OP:$D		`store` *var a*
2OP:$E		`insert_obj` *obj1 obj2*
2OP:$F		`loadw` *baddr n <result>*
2OP:$10		`loadb` *baddr n <result>*
2OP:$11		`get_prop` *obj prop <result>*
2OP:$12		`get_prop_addr` *obj prop <result>*
2OP:$13		`get_next_prop` *obj prop <result>*
2OP:$14		`add` *a b <result>*
2OP:$15		`sub` *a b <result>*
2OP:$16		`mul` *a b <result>*
2OP:$17		`div` *a b <result>*
2OP:$18		`mod` *a b <result>*
2OP:$19	4+	`call_f1` *raddr a1 <result>*
2OP:$1A	5+	`call_p1` *raddr a1*
2OP:$1B	5+	`set_colour` *f b*
2OP:$1C	5+ (–)	`throw` *a fp*
2OP:$1D	—	
2OP:$1E	—	
2OP:$1F	—	

VAR:$0		`call_fv` *raddr [a1 a2 a3] <result>*
VAR:$1		`storew` *baddr n a*
VAR:$2		`storeb` *baddr n byte*
VAR:$3		`put_prop` *obj prop a*
VAR:$4	1-3	`read` *baddr1 baddr2*
	4	`read` *baddr1 baddr2 [time raddr]*
	5+	`read` *baddr1 baddr2 [time raddr] <result>*
VAR:$5		`print_char` *n*
VAR:$6		`print_num` *s*
VAR:$7		`random` *s <result>*
VAR:$8		`push` *a*
VAR:$9	1-5	`pull` *var*
	6+ (–)	`pull` *[baddr] var*
VAR:$A	3+	`split_screen` *n*
VAR:$B	3+	`set_window` *window*
VAR:$C	4+	`call_fd` *raddr [a1 a2 a3 a4 a5 a6 a7] <result>*
VAR:$D	4+	`erase_window` *window*
VAR:$E	4-5	`erase_line`
	6	`erase_line` *[n]*
VAR:$F	4-5	`set_cursor` *s x*
	6	`set_cursor` *s x [window]*
VAR:$10	4+ (–)	`get_cursor` *baddr*
VAR:$11	4+	`set_text_style` *n*
VAR:$12	4+	`buffer_mode` *bit*
VAR:$13	3-4	`output_stream` *s*
	5	`output_stream` *s [baddr]*
	6	`output_stream` *s [baddr w]*
VAR:$14	3+	`input_stream` *n*
VAR:$15	3+	`sound` *n [op time raddr]*
VAR:$16	4+	`read_char` *1 [time raddr] <result>*
VAR:$17	4+	`scan_table` *a baddr n [byte] <result> <branch>*
VAR:$18	5+ (–)	`not` *a <result>*
VAR:$19	5+	`call_pv` *raddr [a1 a2 a3]*
VAR:$1A	5+	`call_pd` *raddr [a1 a2 a3 a4 a5 a6 a7]*
VAR:$1B	5+	`tokenise` *baddr1 baddr2 [baddr3 bit]*
VAR:$1C	5+	`encode_text` *baddr1 p n baddr2*
VAR:$1D	5+	`copy_table` *baddr1 baddr2 a*
VAR:$1E	5+	`print_table` *baddr x [y n]*
VAR:$1F	5+	`check_arg_count` *n <branch>*
EXT:$0	5+	`save` *[baddr1 n baddr2] <result>*
EXT:$1	5+	`restore` *[baddr1 n baddr2] <result>*
EXT:$2	5+	`log_shift` *a s <result>*
EXT:$3	5+ (–)	`art_shift` *a s <result>*
EXT:$4	5	`set_font` *n <result>*
	6	`set_font` *n [window] <result>*

EXT:$5	6	`draw_picture` *pic [y x]*
EXT:$6	6	`picture_data` *pic baddr <branch>*
EXT:$7	6	`erase_picture` *pic [y x]*
EXT:$8	6	`set_margins` *xl xr window*
EXT:$9	5+	`save_undo` *<result>*
EXT:$A	5+	`restore_undo` *<result>*
EXT:$B	—	
EXT:$C	—	
EXT:$D	—	
EXT:$E	—	
EXT:$F	—	
EXT:$10	6	`move_window` *window y x*
EXT:$11	6	`window_size` *window y x*
EXT:$12	6	`window_style` *window flags op*
EXT:$13	6	`get_wind_prop` *window p <result>*
EXT:$14	6	`scroll_window` *window s*
EXT:$15	6	`pop_stack` *n [baddr]*
EXT:$16	6	`read_mouse` *baddr*
EXT:$17	6	`mouse_window` *window*
EXT:$18	6	`push_stack` *a baddr <branch>*
EXT:$19	6	`put_wind_prop` *window p a*
EXT:$1A	6	`print_form` *baddr*
EXT:$1B	6	`make_menu` *n baddr <branch>*
EXT:$1C	6	`picture_table` *baddr*
EXT:$1D	—	
…	…	
EXT:$FF	—	

4. Equivalent mnemonics

This table contains mnemonics used in a number of sources, and their equivalent in this document.[1]

Table 8. Equivalent mnemonics

Mnemonic	Equiv. to	Remarks
`aread`	`read`	Inform name for V5+ `read`
`beep`	`sound`	Inform name for `sound` without operands
`call`	`call_fv`	Inform name for V1-3 `call_fv`
`call_1n`	`call_p0`	
`call_1s`	`call_f0`	
`call_2n`	`call_p1`	

[1] It is tempting to change more mnemonics to better reflect the operations; I try to resist as long as possible. New mnemonics should be easy to understand for someone familiar with old ones.

`call_2s`	`call_f1`	
`call_vn`	`call_pv`	
`call_vs`	`call_fv`	
`call_vn2`	`call_pd`	
`call_vs2`	`call_fd`	
`dec_chk`	`dec_jl`	
`icall`	`call_fv`	Inform name for an ‘indirect call’
`inc_chk`	`inc_jg`	
`print_ret`	`print_rtrue`	
`ret_popped`	`ret_pulled`	
`sound_effect`	`sound`	Inform name for `sound` with operands
`split_window`	`split_screen`	
`sread`	`read`	Inform name for V1-4 `read`
`vje`	`je`	Inform name for variable instr. form

5. Special characters

This table describes what characters 155-223 should look like in an ASCII font. Note that characters 164-223 are Standard extensions.

[**TODO**: Insert table of German characters 155-163, and Standard 0.2 (plus Z-letter #1) characters 164-223; also include suggested multi-character equivalents.]

References

[Frotz] Stefan Jokisch `<jokisch@ls7.informatik.uni_dortmund.de>`. Frotz. One of the most accurate Z-machine emulators currently available, and a spin-off from ZIP. Current version 2.01 runs V1-8 Z-programs under MS-DOS, and a Unix version runs all except V6. Accurate and with many useful features. Available from the `infocom/interpreters` directory of the IF archive.

[IFA] The Interactive Fiction Archive. URL ftp:// ftp.gmd.de/ if-archive. Maintained by Volker Blasius `<blasius@gmd.de>`.

[Inform] Graham Nelson `<nelson@vax.ox.ac.uk>`. Inform. The compiler that produces Z-programs, and comes with a comprehensive library to build your own text adventures with. Current compiler version 6.05, current library version 6/2. Available from the `infocom/compilers` directory of the IF archive.

[Nelson] Graham Nelson `<nelson@vax.ox.ac.uk>`. The Specification of the Z-machine and Inform assembly language (Standard 0.2: 15th November 1995). TEX source and plain ASCII version are in the `infocom/interpreters` directory of the IF archive. This is a consultation version, intended as a prelude to standard 1.0. The “Informal Z-machine Newsletter #1” (10th December 1995, circulated by Graham Nelson) contains a number of corrections and further remarks, and recently Stefan Jokisch also published a list of remarks; both can be found in the above mentioned directory.

[ZIP] Mark Howell `<howell_ma@movies.enet.dec.com>`. ZIP. An accurate Z-machine

emulator for V1-5, which runs on a large range of computer systems, including IBM PC, Amiga and Unix. The most well-known version is version 2.0, dated 10 March 1993. From August 1995 maintained by David Rose `<drose@freenet.tlh.fl.us>`. This version and a number of derivatives are in the `infocom/interpreters` directory of the IF archive.