The Z-Machine Standards Document

Version 1.0

22nd June 1997

two misprints corrected, 9th August

resources appendix updated and discovery added to header table, 4th September

---

Preface

The Z-machine was created on a coffee table in Pittsburgh in 1979. It is an imaginary computer whose programs are adventure games, and is well-adapted to its task, implementing complex games remarkably compactly. They were still perhaps 100K long, too large for the memory of the home computers of their day, and the Z-machine seems to have made the first usage of virtual memory on a microcomputer. Further ahead of its time was the ability to efficiently save and restore the entire execution state.

The design's cardinal principle is that any game is 100% portable to different computers: that is, any legal program exactly determines its behaviour. This portability is largely made possible by a willingness to constrain maximum as well as minimum levels of performance (for instance, dynamic memory allocation is impossible).

Infocom's catalogue continues to be sold and to be played under interpreter programs, either original Infocom ones or more recent and generally better freeware ones. About 130 story files compiled by Infocom's compiler **Zilch** survive and since 1993 very many more story files have been created with the Inform design system.

Eight Versions of the Z-machine exist, and the first byte of any "story file" (that is: any Z-machine program) gives the Version number it must be interpreted under.

Standardisation

The opcode names used in this document were agreed between 1994 and 1995 as a standard set by Mark Howell, author of the disassembler **Txd** (part of the **Ztools** suite of utility programs), and Graham Nelson, author of the assembly level of Inform. They do not correspond to Infocom's unpublished opcode names.

This Standard was drawn up in November 1995, drawing on a rougher description written in 1993 and, before that, sketches of table formats by Mike Threepoint and others. It has formalised what different interpreter writers regard as the Z-machine, guaranteeing a reliable and well-featured platform for writers of new games. The first formal Standard was numbered 0.2, and this is the second, containing some corrections and clarifications but also two new features. The following changes are worth noting:

- Support for the Unicode character set has been added, introducing a new table and two new opcodes. **S**3 has been rewritten and there are also changes to **S**7 and **S**10, as well as the addition of the opcodes to **S**14 and **S**15.

- **S**8.8.3.1, on window attributes in Version 6, has been rewritten with extensive corrections.
- **S**7.1.2.1.1 requires a new feature: handling nested usages of output stream 3.
- It is now explicit that text buffering never applies to the upper window in Versions 3 to 5. (In Standard 0.2 the rules allowed text buffering in Version 4 under some conditions.)
- An optional operand (never used and not useful) has been removed from the opcode **set_font**. An optional operand, discovered by Mark Knibbs, has however been added to the Version 6 form of **set_colour**.
- It is now defined that the input character codes for return and delete are 13 and 8 respectively. (10 and 127 have been suggested as alternatives in the past).
- The fixed-pitch font flag now survives restarts and restores, like the transcription flag.

Also, the "character set table" is now called the "alphabet table" (for clarity) and the "mouse data table" has been renamed the "header extension table."

A companion document to this one, by Martin Frost, defines a standard format called **Quetzal** for saved-game files. Standard interpreters are not *required* to use **Quetzal**, since choice of saved-game format does not affect Z-Machine behaviour, but interpreter-writers are strongly encouraged to consider it.

Andrew Plotkin is currently (June 1997) drafting a standard format called **Blorb** for a "resources" file to accompany or encapsulate a Z-machine game, neatly packaging up sound and graphics in modern formats. Again, since the Z-Machine has no formal knowledge of the means of storage of sound or graphics, this document does not include Andrew's. A Standard Version-6 interpreter need not provide for **Blorb**.

So what is "standard"?

To call itself "Standard", an interpreter should (as far as anyone knows) obey this document exactly for every Version of the Z-machine it claims to interpret. Interpreters need not provide optional features suggested in the "remarks" sections, and need not make their source code public. Each edition of this document has a Revision number, somewhat like the JFIF identification number used by the JPEG standard. A standard interpreter should communicate its revision number in three ways:

- To someone downloading it from an FTP site or bulletin board: by including it in its filename.
- To the player: for instance by means of an "information" option on a menu, or in an initialisation sequence.
- To the game: by writing it into bytes in the header which were always left zero before this standard was devised (see **S**11). A game compiled with Inform library 5/12 or later prints the revision number in its banner (if this isn't 0.0).

Few arbitrary choices have been made in writing this document. Where Infocom's own shipped interpreters disagree, or contain manifest bugs, it has usually been possible to decide which was "correct". Elsewhere, minimum levels of performance have been invented where necessary. (For example, a minimum call-stack size is needed for programmers to be sure of what level of recursion is safe.)

Those few paragraphs which genuinely extend the Infocom format are marked *******. In any event, Infocom's original shipped interpreters do not conform to this standard document, because of bugs or because of slight variations between the Inform output format and Infocom's.

Notation

Hexadecimal numbers are written with an initial dollar, as in **$ff**, while binary numbers are written with a double-dollar as in **$$11011**, according to Inform conventions. The bits in a byte are numbered 0 to 7, 0 being the least significant and the top bit, 7, the most.

Story files are mechanically best identified by their release number and serial code, which are written into the header information at the bottom of Z-machine memory. The release number can be anything between 0 and 65535 but is usually between 1 and 100. The serial code can consist of any six textual characters but is usually the date of compilation, arranged **YYMMDD**: thus 970619 refers to June 19th, 1997.

Paul David Doherty, in his extensive investigations into Infocom's released games, introduced the notation

Release number.Serial code

to identify particular story files: for example the first production copy of 'Enchanter' is 10.830810. This notation is used throughout the Standard when individual Infocom files need to be referred to.

Where are all the grammar tables?

The Z-machine has some lexical acuity but it doesn't contain a full parser: it's like a computer without an operating system. A game program has to contain its own parser and the tables this uses are not part of the formal Z-machine specification. (Many Infocom games have similar parsing table formats simply because, until Version 6, they used an evolving version of the 'Zork I' parser. A quite different parser was used in Version 6.) Inform's parsing table formats are documented in the *Inform Technical Manual*. For the usual format of Infocom's parsing tables, see the **Ztools** utility **Infodump**.

Acknowledgements

> There is an obvious resemblance between an unreadable script and a secret code; similar methods can be employed to break both. But the differences must not be overlooked. The code is deliberately designed to baffle the investigator; the script is only puzzling by accident.
>
> John Chadwick, **The Decipherment of Linear B**

The Z-machine was originally devised by Joel Berez and Marc Blank in 1979. Marc Blank made most of the Version 4 extensions, and Version 5 was created by Dave Lebling (with contributions from others including Brian Moriarty, Duncan Blanchard and Linde Dynneson). Version 6 was largely the work of Tim Anderson and Dave Lebling.

In the reverse direction, decipherment is mostly due to the InfoTaskForce (David Beazley, George Janczuk, Peter Lisle, Russell Hoare and Chris Tham), Matthias Pfaller, Mike Threepoint, Mark Howell, Paul David Doherty and Stefan Jokisch. Only a few of the pieces in the jigsaw were placed by myself.

I gratefully acknowledge the help of Paul David Doherty and Mark Howell, who each read drafts of this paper and sent back detailed corrections; also, of Stefan Jokisch and Marnix Klooster who have put a great deal of work into the fine detail of the specification; and of all those who commented on the circulated draft. Mistakes and misunderstandings remain my own.

Graham Nelson

15 November 1995

Kevin Bracey and Stefan Jokisch discovered most of the mistakes in Standard 0.2, in developing the first Version 6 interpreters of the modern age: **Zip2000** and **Frotz**. Matthew Russotto and Mark Knibbs supplied helpful information about Infocom's own Version 6 interpreters. Stefan also kindly read and commented on numerous drafts of the present revision. Finally, discussion about this document was greatly assisted by the Z-Machine Mailing List, organised by Marnix Klooster.

Graham Nelson

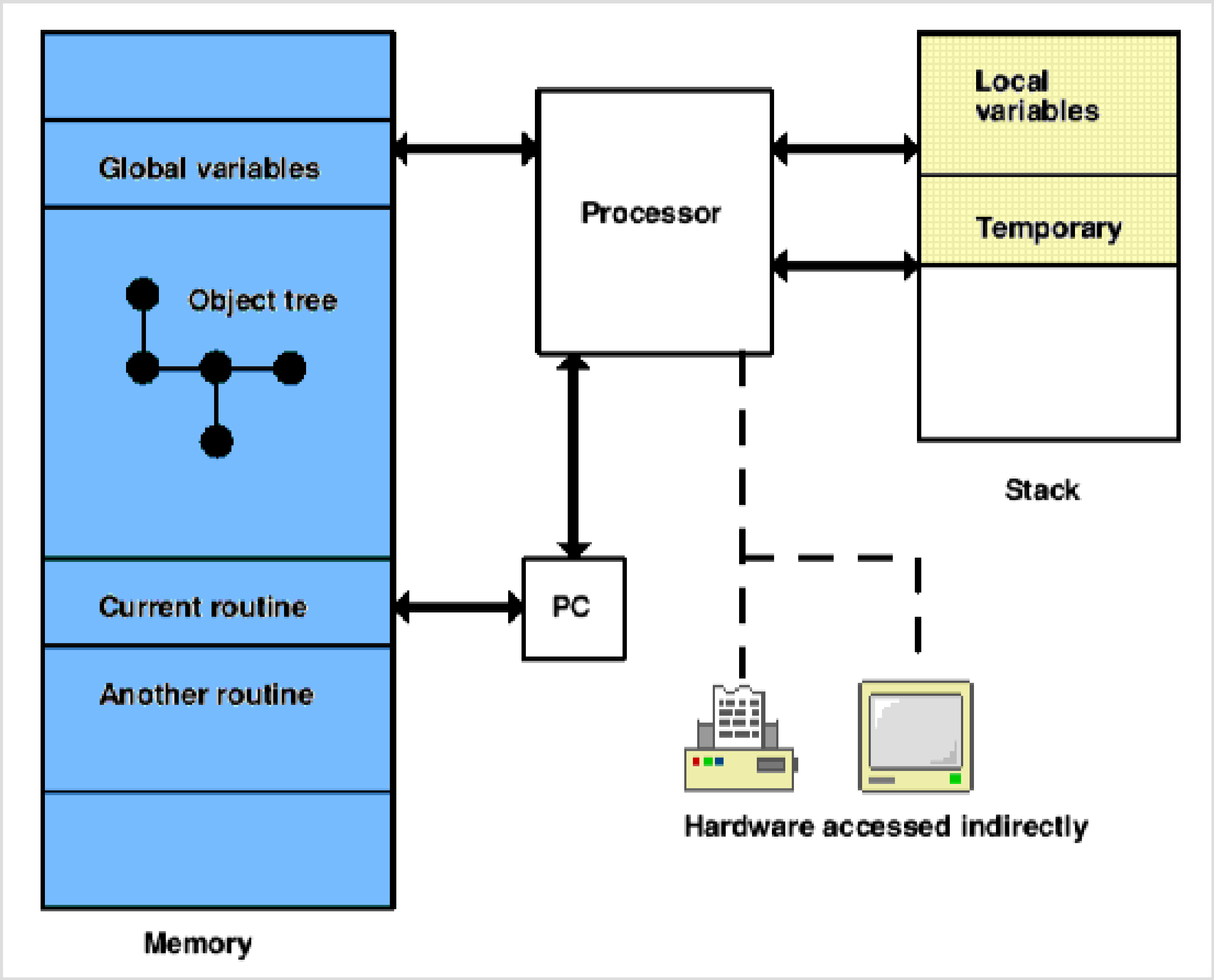
22 June 1997

Contents / Preface / Overview

Section 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9 / 10 / 11 / 12 / 13 / 14 / 15 / 16

Appendix A / B / C / D / E / F

Overview of Z-machine architecture



The Z-machine is a design for an imaginary computer: Z is for 'Zork', the adventure game it was originally designed to play. Like any computer, it stores its information (mostly) in an array of variables numbered from 0 up to some large number: this is called its **memory**. A stock of some 240 memory locations are set aside for easy and quick access, and these are called **global variables** (since they are available to any part of the program which is running, at any time).

The two important pieces of information not stored in memory are the **program counter** (PC) and the **stack**. The Z-machine continuously runs a program by getting the instruction stored at position PC in memory, acting on the instruction and then moving the PC forward to the next. The **instruction set** of the Z-machine (the range of possible actions and how they are encoded as numbers in memory) occupies much of this document.

Programs are divided into **routines**: the Z-machine is always executing a particular routine, the one which the PC currently points inside. However, some instructions cause the Z-machine to **call** a new routine and then to return where the first routine left off. The Z-machine therefore needs to remember details of where to go back, and it stores these on the stack.

The stack is a second bank of memory, quite separate from the main one, which has variable size: initially it is empty. From time to time values are added to, or taken from, the top of the stack. As well as being used to keep return details, the stack is also used to store **local variables** (values needed only by a particular routine) and, for short periods only, the partial results of calculations.

Thus, whereas most physical processors (e.g. Z80 or 6502) have a number of quick-access variables outside of memory (called "registers") and a stack inside memory, the Z-machine has the reverse: it has global variables inside memory and a stack kept outside.

There is no access to hardware except by executing particular Z-machine instructions. For instance, **read** and **read_char** allow use of the keyboard; **print** and **draw_picture** allow use of the screen. The screen's image is not stored anywhere in memory. Conversely, hardware can cause the Z-machine to **interrupt**, that is, to make a spontaneous call to a particular routine, interrupting what it was previously working on. This happens only if the program has previously requested it: for example, by setting a sound effect playing and asking for a routine to be called when it finishes; or by asking for an interrupt if thirty seconds pass while the player is thinking what to type.

This simple architecture is overlaid by a number of special structures which the Z-machine maintains inside memory. There are around a dozen of these but the most important are:

the **header**, at the bottom of memory, giving details about the program and a map of the rest of memory;

the **dictionary**, a list of English words which the program expects that it might want to read from the keyboard;

the **object tree**, an arrangement of chunks of memory called **objects**.

The Z-machine is primarily used for adventure games, where the dictionary holds names of items and verbs that the player might type, and the objects tend to be the places and artifacts which make up the game. Each object in the tree may have a **parent**, a **sibling** and a **child**. For instance, in the start position of 'Zork I':

> **West of House**
>
> You are standing in an open field west of a white house, with a boarded front door. There is a small mailbox here.
>
> >open mailbox
>
> Opening the small mailbox reveals a leaflet.

At this point (part of) the game's object tree looks like this:

```
[ 41] ""
 . [ 68] "West of House"
 .  . [ 21] "you"
 .  . [239] "small mailbox"
 .  .  . [ 80] "leaflet"
 .  . [127] "door"
```

Note that objects are numbered from 1 upward. (Object 41 is a dummy object being used by the game to contain all the "rooms" or locations, and it has many more children besides object 68.) The parent of the player is "West of House", whose parent is 41, which has no parent. The sibling of the player is the mailbox; the child of the mailbox is the leaflet; the sibling of the mailbox is the door and so on.

Objects are bundled-up collections of variables, which come in two kinds: **attributes** and **properties**. Attributes are simply flags, that is, they can be set or unset, but have no numerical value. Properties hold numbers, which may in turn represent pieces of text or other information. For instance, one of the properties of the mailbox object above contains the information that the English word "mailbox" refers to it. One of the attributes of the mailbox object is set to indicate that it's a container, whereas the same attribute for the leaflet object is unset. Here is a breakdown of the state of the mailbox:

```
239. Attributes: 30, 34
     Parent object:  68  Sibling object: 127  Child object:  80
     Property address: 2b53
         Description: "small mailbox"
          Properties:
              [49] 00 0a
              [46] 54 bf 4a c3
              [45] 3e c1
              [44] 5b 1c
```

So the only set attributes are 30 and 34: all others are unset. Values are given for properties 44, 45, 46 and 49. The Z-machine itself does not know or care what this information means: that is for the program to sort out.

As a final example, here is part of one of the routines in 'Zork I':

```
l0006: print_ret       "Suicide is not the answer."
l0007: je              g57 #84 ~l0008
       je              g48 #15 ~rfalse
       print_ret       "Why don't you just walk like normal people?"
l0008: je              g57 #63 ~l0009
       print_ret       "How romantic!"
l0009: je              g57 #3b ~rfalse
       get_parent      "mirror" local0
       get_parent      "mirror" sp
       je              g6b local0 sp ~l0010
       print_ret       "Your image in the mirror looks tired."
l0010: print_ret       "That's difficult unless your eyes are prehensile."
```

Z-machine programs are stored on disc, or archived on the Internet, in what are called **story files**. (Since they were introduced to hold interactive stories.) A story file consists of a snapshot of main memory only. The processor begins to run a story file by starting with an empty stack and a PC value set according to some information in the story file's header. Note that the story file has to be set up with many of the structures in memory, such as the dictionary and the object tree, already created and with sensible contents.

The first byte of any story file, and so the byte at memory address 0, always contains the **version number** of the Z-machine to be used. The design was evolutionary over a period of a decade: as version number increases, the instruction set grows and tables are reformatted to allow more room for larger games. All of Infocom's games can be played using versions between 3 (the majority) and 6. Games compiled by Inform in the 1990s mainly use versions 5 or 8.

Contents / Preface / Overview

Section 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9 / 10 / 11 / 12 / 13 / 14 / 15 / 16

Appendix A / B / C / D / E / F

---

1. The memory map

1.1

The memory map of the Z-machine is an array of bytes with "byte addresses" running from 0 upwards. This is divided into three regions: "dynamic", "static" and "high". Dynamic memory begins from byte address **$00000** and runs up to the byte before the byte address stored in the word at **$0e** in the header. (Dynamic memory must contain at least 64 bytes.) Static memory follows immediately on. Its extent is not defined in the header (or anywhere else), though it must end by the last byte of the story file or by byte address **$0ffff** (whichever is lower). High memory begins at the "high memory mark" (the byte address stored in the word at **$04** in the header) and continues to the end of the story file. The bottom of high memory may overlap with the top of static memory (but not with dynamic memory).

1.1.1

Dynamic memory can be read or written to (either directly, using **loadb**, **loadw**, **storeb** and **storew**, or indirectly with opcodes such as **insert_obj** and **remove_obj**).

1.1.1.1

By tradition, the first 64 bytes are known as the "header". The contents of this are given later but note that games are not permitted to alter many bits inside it.

1.1.1.2

It is legal for games to alter any of the tables stored in dynamic memory above the header, provided they leave the tables in legal states.

1.1.2

Static memory can be read using the opcodes **loadb** and **loadw**. It is illegal for a game to attempt to write to static memory.

1.1.3

Except for its (possible) overlap with static memory, high memory cannot be directly accessed at all by a game program. It contains routines, which can be called, and strings, which can be printed using **print_paddr**.

1.1.4

The maximum permitted length of a story file depends on the Version, as follows:

```
    V1-3    V4-5    V6      V7      V8
    128     256     512     320     512
```

1.2

There are three kinds of address in the Z-machine, all of which can be stored in a 2-byte number: byte addresses, word addresses and packed addresses.

1.2.1

A byte address specifies a byte in memory in the range 0 up to the last byte of static memory.

1.2.2

A word address specifies an even address in the bottom 128K of memory (by giving the address divided by 2). (Word addresses are used only in the abbreviations table.)

1.2.3

******* A packed address specifies where a routine or string begins in high memory. Given a packed address *P*, the formula to obtain the corresponding byte address *B* is:

```
  2P           Versions 1, 2 and 3
  4P           Versions 4 and 5
  4P + 8R_O    Versions 6 and 7, for routine calls
  4P + 8S_O    Versions 6 and 7, for print_paddr
  8P           Version 8
```

R_O and *S_O* are the routine and strings offsets (specified in the header as words at **$28** and **$2a**, respectively).

An example memory map of a small game

Dynamic	**00000**	header
	00040	abbreviation strings

	00042	abbreviation table
	00102	property defaults
	00140	objects
	002f0	object descriptions and properties
	006e3	global variables
	008c3	arrays
Static	**00b48**	grammar table
	010a7	actions table
	01153	preactions table
	01201	adjectives table
	0124d	dictionary
High	**01a0a**	Z-code
	05d56	static strings
	06ae6	end of file

Remarks

Inform never compiles any overlap between static and high memory (it places all data tables in dynamic memory). However, many Infocom games group tables of static data just above the high memory mark, before routines begin; some, such as 'Nord 'n' Bert...', interleave static data between routines, so that static memory actually overlaps code; and a few, such as 'Seastalker' release 15, even contain routines placed below the high memory mark. (The original idea behind the high memory mark was that everything below it should be stored in the interpreter's RAM, while what was above could reasonably be kept in "virtual memory", i.e., loaded off disc as needed.)

Note that the total of dynamic plus static memory must not exceed 64K. (In fact, 64K minus 2 bytes.) This is the most serious limitation on the Z-machine (though it has not yet been reached by anyone).

Throughout the specification, Versions 7 and 8 are identical to Version 5 except as stated at 1.1.4 and 1.2.3 above.

Contents / Preface / Overview

Section 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9 / 10 / 11 / 12 / 13 / 14 / 15 / 16

Appendix A / B / C / D / E / F

---

2. Numbers and arithmetic

2.1

In the Z-machine, numbers are usually stored in 2 bytes (in the form most-significant-byte first, then least-significant) and hold any value in the range **$0000** to **$ffff** (0 to 65535 decimal).

2.2

These values are sometimes regarded as signed, in the range -32768 to 32767. In effect $-n$ is stored as $65536-n$ and so the top bit is the sign bit.

2.2.1

The operations of numerical comparison, multiplication, addition, subtraction, division, remainder-after-division and printing of numbers are signed; bitwise operations are unsigned. (In particular, since comparison is signed, it is unsafe to compare two addresses using simply **jl** and **jg**.)

2.3

Arithmetic errors:

2.3.1

It is illegal to divide by 0 (or to ask for remainder after division by 0) and an interpreter should halt with an error message if this occurs.

2.3.2

Formally it has never been specified what the result of an out-of-range calculation should be. The author suggests that the result should be reduced modulo **$10000**.

2.4

The Z-machine needs a random number generator which at any time has one of two states, "random" and "predictable". When the game starts or restarts the state becomes "random". Ideally the generator should not produce identical sequences after each restart.

2.4.1

When "random", it must be capable of generating a uniformly random integer in the range 1 <= x <= n, for any value 1 <= n <= 32767. Any method can be used for this (for instance, using the host computer's clock time in milliseconds). The uniformity of randomness should be optimised for low values of n (say, up to 100 or so) and it is especially important to avoid regular patterns appearing in remainders after division (most crudely, being alternately odd and even).

2.4.2

The generator is switched into "predictable" state with a seed value. On any two occasions when the same seed is sown, identical sequences of values must result (for an indefinite period) until the generator is switched back into "random" mode. The generator should cope well with very low seed values, such as 10, and should not depend on the seed containing many non-zero bits.

2.4.3

The interpreter is permitted to switch between these states on request of the player. (This is useful for testing purposes.)

Remarks

It is dangerous to rely on the ANSI C random number routines, as some implementations of these are very poor. This has made some games (in particular, 'Balances') unwinnable on some Unix ports of **Zip**.

The author suggests the following algorithm:

1. In "random" mode, the generator uses the host computer's clock to obtain a random sequence of bits.

2. In "predictable" mode, the generator should store the seed value *S*. If *S* < 1000 it should then internally generate

1, 2, 3, ..., *S*, 1, 2, 3, ..., *S*, 1, ...

so that **random n** produces the next entry in this sequence modulo n. If *S* >= 1000 then *S* is used as a seed in a standard seeded random-number generator.

(The rising sequence is useful for testing, since it will produce all possible values in sequence. On the other hand, a seeded but fairly random generator is useful for testing entire scripts.)

Note that version 0.2 of this standard mistakenly asserted that division and remainder are unsigned, a myth deriving from a bug in **Zip**. Infocom's interpreters do sign division (this is relied on when calculating pizza cooking times for the microwave oven in 'The Lurking Horror'). Here are some correct Z-machine calculations:

```
   -11 / 2 = -5         -11 / -2 = 5          11 / -2 = -5
   -13 % 5 = -3          13 % -5 = 3         -13 % -5 = -3
```

---

3. How text and characters are encoded

This technique is similar to the five-bit Baudot code, which was used by early Teletypes before ASCII was invented.

Marc S. Blank and S. W. Galley, **How to Fit a Large Program Into a Small Machine**

3.1

Z-machine text is a sequence of ZSCII character codes (ZSCII is a system similar to ASCII: see **S** 3.8 below). These ZSCII values are encoded into memory using a string of Z-characters. The process of converting between Z-characters and ZSCII values is given in **SS** 3.2 to 3.7 below.

3.2

Text in memory consists of a sequence of 2-byte words. Each word is divided into three 5-bit 'Z-characters', plus 1 bit left over, arranged as

```
   --first byte-------   --second byte---
   7    6 5 4 3 2  1 0   7 6 5  4 3 2 1 0
   bit  --first--  --second---  --third--
```

The bit is set only on the last 2-byte word of the text, and so marks the end.

3.2.1

There are three 'alphabets', A0 (lower case), A1 (upper case) and A2 (punctuation) and during printing one of these is current at any given time. Initially A0 is current. The meaning of a Z-character may depend on which alphabet is current.

3.2.2

In Versions 1 and 2, the current alphabet can be any of the three. The Z-characters 2 and 3 are called 'shift' characters and change the alphabet for the next character only. The new alphabet depends on what the current one is:

```
             from A0  from A1  from A2
  Z-char 2      A1       A2       A0
  Z-char 3      A2       A0       A1
```

Z-characters 4 and 5 permanently change alphabet, according to the same table, and are called 'shift lock' characters.

3.2.3

In Versions 3 and later, the current alphabet is always A0 unless changed for 1 character only: Z-characters 4 and 5 are shift characters. Thus 4 means "the next character is in A1" and 5 means "the next is in A2". There are no shift lock characters.

3.2.4

An indefinite sequence of shift or shift lock characters is legal (but prints nothing).

3.3

In Versions 3 and later, Z-characters 1, 2 and 3 represent abbreviations, sometimes also called 'synonyms' (for traditional reasons): the next Z-character indicates which abbreviation string to print. If z is the first Z-character (1, 2 or 3) and x the subsequent one, then the interpreter must look up entry 32(z-1)+x in the abbreviations table and print the string at that word address. In Version 2, Z-character 1 has this effect (but 2 and 3 do not, so there are only 32 abbreviations).

3.3.1

Abbreviation string-printing follows all the rules of this section except that an abbreviation string must not itself use abbreviations and must not end with an incomplete multi-Z-character construction (see **S** 3.6.1 below).

3.4

Z-character 6 from A2 means that the two subsequent Z-characters specify a ten-bit ZSCII character code: the next Z-character gives the top 5 bits and the one after the bottom 5.

3.5

The remaining Z-characters are translated into ZSCII character codes using the "alphabet table".

3.5.1

The Z-character 0 is printed as a space (ZSCII 32).

3.5.2

In Version 1, Z-character 1 is printed as a new-line (ZSCII 13).

3.5.3

In Versions 2 to 4, the alphabet table for converting Z-characters into ZSCII character codes is as follows:

```
   Z-char 6789abcdef0123456789abcdef
current   --------------------------
  A0      abcdefghijklmnopqrstuvwxyz
  A1      ABCDEFGHIJKLMNOPQRSTUVWXYZ
  A2       ^0123456789.,!?_#'"/\-:()
          --------------------------
```

(Character 6 in A2 is printed as a space here, but is not translated using the alphabet table: see **S** 3.4 above. Character 7 in A2, written here as a circumflex ^, is a new-line.) For example, in alphabet A1 the Z-character 12 is translated as a capital G (ZSCII character code 71).

3.5.4

Version 1 has a slightly different A2 row in its alphabet table (new-line is not needed, making room for the < character):

```
          6789abcdef0123456789abcdef
          --------------------------
  A2       0123456789.,!?_#'"/\<-:()
          --------------------------
```

3.5.5

In Versions 5 and later, the interpreter should look at the word at **$34** in the header. If this is zero, then the alphabet table drawn out in **S** 3.5.3 continues in use. Otherwise it is interpreted as the byte address of an alphabet table specific to this story file.

3.5.5.1

Such an alphabet table consists of 78 bytes arranged as 3 blocks of 26 ZSCII values, translating Z-characters 6 to 31 for alphabets A0, A1 and A2. Z-characters 6 and 7 of A2, however, are still translated as escape and newline codes (as above).

3.6

Since the end-bit only comes up once every three Z-characters, a string may have to be 'padded out' with null values. This is conventionally achieved with a sequence of 5's, though a sequence of (for example) 4's would work equally well.

3.6.1

It is legal for the string to end while a multi-Z-character construction is incomplete: for instance, after only the top half of an ASCII value has been given. The partial construction is simply ignored. (This can happen in printing dictionary words which have been guillotined to the dictionary resolution of 6 or 9 Z-characters.)

3.7

When an interpreter is encrypting typed-in text to match against dictionary words, the following restrictions apply. Text should be converted to lower case (as a result A1 will not be needed unless the game provides its own alphabet table). Abbreviations may not be used. The pad character, if needed, must be 5. The total string length must be 6 Z-characters (in Versions 1 to 3) or 9 (Versions 4 and later): any multi-Z-character constructions should be left incomplete (rather than omitted) if there's no room to finish them. For example, "i" is encrypted as:

14, 5, 5, 5, 5, 5, 5, 5, 5

$48a5 $14a5 $94a5

3.8

The character set of the Z-machine is called ZSCII (Zork Standard Code for Information Interchange; pronounced to rhyme with "xyzzy"). ZSCII codes are 10-bit unsigned values between 0 and 1023. Story files may only legally use the values which are defined below. Note that some values are defined only for input and some only for output.

Table 2: summary of the ZSCII rules

0	null	Output
1-7	----	
8	delete	Input
9	tab (V6)	Output
10	----	
11	sentence space (V6)	Output
12	----	
13	newline	Input/Output
14-26	----	
27	escape	Input

28-31	----	
32-126	standard ASCII	Input/Output
127-128	----	
129-132	cursor u/d/l/r	Input
133-144	function keys f1 to f12	Input
145-154	keypad 0 to 9	Input
155-251	extra characters	Input/Output
252	menu click (V6)	Input
253	double-click (V6)	Input
254	single-click	Input
255-1023	----	

3.8.1

The codes 256 to 1023 are undefined, so that for all practical purposes ZSCII is an 8-bit unsigned code.

3.8.2

The codes 0 to 31 are undefined except as follows:

3.8.2.1

ZSCII code 0 ("null") is defined for output but has no effect in any output stream. (It is also used as a value meaning "no character" when reporting terminating character codes, but is not formally defined for input.)

3.8.2.2

ZSCII code 8 ("delete") is defined for input only.

3.8.2.3

ZSCII code 9 ("tab") is defined for output in Version 6 only. At the start of a screen line this should print a paragraph indentation suitable for the font being used: if it is printed in the middle of a screen line, it should be converted to a space (Infocom's own interpreters do not do this, however).

3.8.2.4

ZSCII code 11 ("sentence space") is defined for output in Version 6 only. This should be printed as a suitable gap between two sentences (in the same way that typographers normally place larger spaces after the full stops ending sentences than after words or commas).

3.8.2.5

ZSCII code 13 ("carriage return") is defined for input and output.

3.8.2.6

ZSCII code 27 ("escape" or "break") is defined for input only.

3.8.3

ZSCII codes between 32 ("space") and 126 ("tilde") are defined for input and output, and agree with standard ASCII (as well as all of the ISO 8859 character sets and Unicode). Specifically:

```
      0123456789abcdef0123456789abcdef
      --------------------------------
 $20   !"#$%&'()*+,-./0123456789:;<=>?
 $40  @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
 $60  'abcdefghijklmnopqrstuvwxyz{!}~
      --------------------------------
```

Note that code **$23** (35 decimal) is a hash mark, not a pound sign. (Code **$7c** (124 decimal) is a vertical stroke which is shown as **!** here for typesetting reasons.)

3.8.3.1

ZSCII codes 127 ("delete" in some forms of ASCII) and 128 are undefined.

3.8.4

ZSCII codes 129 to 154 are defined for input only:

```
129: cursor up  130: cursor down  131: cursor left  132: cursor right
133: f1         134: f2           ....              144: f12
145: keypad 0   146: keypad 1     ....              154: keypad 9
```

3.8.5

The block of codes between 155 and 251 are the "extra characters" and are used differently by different story files. Some will need accented Latin characters (such as French E-acute), others unusual punctuation (Spanish question mark), others new alphabets (Cyrillic or Hebrew); still others may want dingbat characters, mathematical or musical symbols, and so on.

3.8.5.1

******* To define which characters are required, the Unicode (or ISO 10646-1) character set is used: characters are specified by unsigned 16-bit codes. These values agree with ISO 8859 Latin-1 in the range 0 to 255, and with ASCII and ZSCII in the range 32 to 126. The Unicode standard leaves a range of values, the Private Use Area, free: however, an Internet group called the ConScript Unicode Registry is organising a standard mapping of invented scripts (such as Klingon, or Tolkien's Elvish) into the Private Use Area, and this should be considered part of the Unicode standard for Z-machine purposes.

3.8.5.2

******* The story file chooses its stock of extra characters with a "Unicode translation table" as follows. Under Versions 1 to 4, the "default table" is always used (see below). In Version 5 or later, if Word 3 of the header extension table is present and non-zero then it is interpreted as the byte address of the Unicode translation table. If Word 3 is absent or zero, the default table is used.

3.8.5.2.1

The table consists of one byte giving a number N, followed by N two-byte words.

3.8.5.2.2

This indicates that ZSCII characters 155 to $155+N-1$ are defined for both input and output. (It's possible for N to be zero, leaving the whole range 155 to 251 undefined.)

3.8.5.2.3

The words in the table give Unicode character codes for each of the ZSCII characters 155 to $155+N-1$ in turn.

3.8.5.3

The default table is as shown in Table 1.

3.8.5.4

The defined extra characters are entirely normal ZSCII characters. They can appear in a story file's alphabet table, in an array created by print stream 3 and so on.

3.8.5.4.1

******* The interpreter is required to be able to print representations of every defined Unicode character under **$0100** (i.e. of every defined ISO 8859-1 Latin1 character). If no suitable letter forms are available, textual equivalents may be used (such as "ss" in place of German sharp "s").

3.8.5.4.2

Normally, and where sensibly possible, all punctuation and letter characters in ISO 8859-1 Latin1 should be readable from the interpreter's keyboard. (However, some interpreters may want to provide alternative keyboard mappings, or to run in a different ISO 8859 set: Cyrillic, for example.)

3.8.5.4.3

******* An interpreter is not required to have suitable letter-forms for printing Unicode characters **$0100** to **$FFFF**. (It may, if it chooses, allow the user to configure certain fonts for certain Unicode ranges; but this is not required.) If a Unicode character must be printed which an interpreter has no letter-form for, a question mark should be printed instead.

3.8.6

ZSCII codes 252 to 254 are defined for input only:

```
252: menu click   253: mouse double-click   254: mouse single-click
```

Menu clicks are available only in Version 6. In Versions 5 and later it is recommended that an interpreter should only send code 254, whether the mouse is clicked once or twice.

3.8.7

ZSCII code 255 is undefined. (This value is needed in the "terminating characters table" as a wildcard, indicating "any Input-only character with code 128 or above." However, it cannot itself be printed or read from the keyboard.)

Table 1: default Unicode translations (see S 3.8.5.3)

155	0e4	a-diaeresis	**ae**	191	0e2	a-circumflex	**a**
156	0f6	o-diaeresis	**oe**	192	0ea	e-circumflex	**e**
157	0fc	u-diaeresis	**ue**	193	0ee	i-circumflex	**i**
158	0c4	A-diaeresis	**Ae**	194	0f4	o-circumflex	**o**
159	0d6	O-diaeresis	**Oe**	195	0fb	u-circumflex	**u**
160	0dc	U-diaeresis	**Ue**	196	0c2	A-circumflex	**A**
161	0df	sz-ligature	**ss**	197	0ca	E-circumflex	**E**
162	0bb	quotation	>> or "	198	0ce	I-circumflex	**I**
163	0ab	marks	<< or "	199	0d4	O-circumflex	**O**
164	0eb	e-diaeresis	**e**	200	0db	U-circumflex	**U**
165	0ef	i-diaeresis	**i**	201	0e5	a-ring	**a**
166	0ff	y-diaeresis	**y**	202	0c5	A-ring	**A**
167	0cb	E-diaeresis	**E**	203	0f8	o-slash	**o**
168	0cf	I-diaeresis	**I**	204	0d8	O-slash	**O**

169	0e1	a-acute	**a**	205	0e3	a-tilde	**a**
170	0e9	e-acute	**e**	206	0f1	n-tilde	**n**
171	0ed	i-acute	**i**	207	0f5	o-tilde	**o**
172	0f3	o-acute	**o**	208	0c3	A-tilde	**A**
173	0fa	u-acute	**u**	209	0d1	N-tilde	**N**
174	0fd	y-acute	**y**	210	0d5	O-tilde	**O**
175	0c1	A-acute	**A**	211	0e6	ae-ligature	**ae**
176	0c9	E-acute	**E**	212	0c6	AE-ligature	**AE**
177	0cd	I-acute	**I**	213	0e7	c-cedilla	**c**
178	0d3	O-acute	**O**	214	0c7	C-cedilla	**C**
179	0da	U-acute	**U**	215	0fe	Icelandic thorn	**th**
180	0dd	Y-acute	**Y**	216	0f0	Icelandic eth	**th**
181	0e0	a-grave	**a**	217	0de	Icelandic Thorn	**Th**
182	0e8	e-grave	**e**	218	0d0	Icelandic Eth	**Th**
183	0ec	i-grave	**i**	219	0a3	pound symbol	**L**
184	0f2	o-grave	**o**	220	153	oe-ligature	**oe**
185	0f9	u-grave	**u**	221	152	OE-ligature	**OE**
186	0c0	A-grave	**A**	222	0a1	inverted !	**!**
187	0c8	E-grave	**E**	223	0bf	inverted ?	**?**
188	0cc	I-grave	**I**				
189	0d2	O-grave	**O**				
190	0d9	U-grave	**U**			N = 69	

Remarks

In practice the text compression factor is not really very good: for instance, 155000 characters of text squashes into 99000 bytes. (Text usually accounts for about 75\% of a story file.) Encoding does at least encrypt the text so that casual browsers can't read it. Well-chosen abbreviations will reduce total story file size by 10\% or so.

The German translation of 'Zork I' uses an alphabet table to make accented letters (from the standard extra characters set) efficient in dictionary words. In Version 6, 'Shogun' also uses an alphabet table.

Unicode translation tables are new in Standard 1.0: in Standard 0.2, the extra characters were always mapped using the default Unicode translation table.

Note that if a random stretch of memory is accidentally printed as a string (due to an error in the story file), illegal ZSCII codes may well be printed using the 4-Z-character escape sequence. It's helpful for interpreters to filter out any such illegal codes so that the resulting on-screen mess will not cause trouble for the terminal (e.g. by causing the interpreter to print ASCII 12, clear screen, or 7, bell sound).

The continental European quotation marks << and >> should have spacing which looks sensible either in French style <<Merci!>> or in German style >>Danke!<<.

Ideally, an interpreter should be able to read time delays (for timed input) from stream 1 (i.e., from a script file). See the remarks in **S** 7.

The 'Beyond Zork' story file is capable of receiving both mouse-click codes (253 and 254), listing both in its terminating characters table and treating them equally.

The extant Infocom games in Versions 4 and 5 use the control characters 1 to 31 only as follows: they all accept 10 or 13 as equivalent, except that 'Bureaucracy' will only accept 13. 'Bureaucracy' needs either 127 or 8 to be a delete code. No other codes are used.

Curiously, 'Nord 'n' Bert Couldn't Make Head Nor Tail Of It' and 'A Mind Forever Voyaging' allow some letter characters to be typed in with the top bit set. That is, if reading an A, they would recognise 65 or 91 (upper or lower case) and also 193 or 219. Matthew Russotto suggests this was an accommodation for the Apple II, whose keyboard primitives returned the last key pressed in the bottom 7 bits of a byte, plus a top bit flag indicating whether or not the keyboard had been hit since last time.

---

4. How instructions are encoded

We do but teach bloody instructions

Which, being taught, return to plague th' inventor

Shakespeare, **Macbeth**

4.1

A single Z-machine instruction consists of the following sections (and in the order shown):

Opcode	1 or 2 bytes
(Types of operands)	1 or 2 bytes: 4 or 8 2-bit fields
Operands	Between 0 and 8 of these: each 1 or 2 bytes
(Store variable)	1 byte
(Branch offset)	1 or 2 bytes
(Text to print)	An encoded string (of unlimited length)

Bracketed sections are not present in all opcodes. (A few opcodes take both "store" and "branch".)

4.2

There are four 'types' of operand. These are often specified by a number stored in 2 binary digits:

$$00	Large constant (0 to 65535)	2 bytes
$$01	Small constant (0 to 255)	1 byte
$$10	Variable	1 byte
$$11	Omitted altogether	0 bytes

4.2.1

Large constants, like all 2-byte words of data in the Z-machine, are stored with most significant byte first (e.g. **$2478** is stored as **$24** followed by **$78**). A 'large constant' may in fact be a small number.

4.2.2

Variable number **$00** refers to the top of the stack, **$01** to **$0f** mean the local variables of the current routine and **$10** to **$ff** mean the global variables. It is illegal to refer to local variables which do not exist for the current routine (there may even be none).

4.2.3

The type 'Variable' really means "variable by value". Some instructions take as an operand a "variable by reference": for instance, **inc** has one operand, the reference number of a variable to increment. This operand usually has type 'Small constant' (and Inform automatically assembles a line like **@inc turns** by writing the operand **turns** as a small constant with value the reference number of the variable **turns**).

4.3

Each instruction has a form (long, short, extended or variable) and an operand count (0OP, 1OP, 2OP or VAR). If the top two bits of the opcode are **$$11** the form is variable; if **$$10**, the form is short. If the opcode is 190 (**$BE** in hexadecimal) and the version is 5 or later, the form is "extended". Otherwise, the form is "long".

4.3.1

In short form, bits 4 and 5 of the opcode byte give an operand type as above. If this is **$11** then the operand count is 0OP; otherwise, 1OP. In either case the opcode number is given in the bottom 4 bits.

4.3.2

In long form the operand count is always 2OP. The opcode number is given in the bottom 5 bits.

4.3.3

In variable form, if bit 5 is 0 then the count is 2OP; if it is 1, then the count is VAR. The opcode number is given in the bottom 5 bits.

4.3.4

In extended form, the operand count is VAR. The opcode number is given in a second opcode byte.

4.4

Next, the types of the operands are specified.

4.4.1

In short form, bits 4 and 5 of the opcode give the type.

4.4.2

In long form, bit 6 of the opcode gives the type of the first operand, bit 5 of the second. A value of 0 means a small constant and 1 means a variable. (If a 2OP instruction needs a large constant as operand, then it should be assembled in variable rather than long form.)

4.4.3

In variable or extended forms, a byte of 4 operand types is given next. This contains 4 2-bit fields: bits 6 and 7 are the first field, bits 0 and 1 the fourth. The values are operand types as above. Once one type has been given as 'omitted', all subsequent ones must be. Example: **$$00101111** means large constant followed by variable (and no third or fourth opcode).

4.4.3.1

In the special case of the "double variable" VAR opcodes **call_vs2** and **call_vn2** (opcode numbers 12 and 26), a second byte of types is given, containing the types for the next four operands.

4.5

The operands are given next. Operand counts of 0OP, 1OP or 2OP require 0, 1 or 2 operands to be given, respectively. If the count is VAR, there must be as many operands as there were types other than 'omitted'.

4.5.1

Note that only **call_vs2** and **call_vn2** can have more than 4 operands, and no instruction can have more than 8.

4.6

"Store" instructions return a value: e.g., **mul** multiplies its two operands together. Such instructions must be followed by a single byte giving the variable number of where to put the result.

4.7

Instructions which test a condition are called "branch" instructions. The branch information is stored in one or two bytes, indicating what to do with the result of the test. If bit 7 of the first byte is 0, a branch occurs when the condition was false; if 1, then branch is on true. If bit 6 is set, then the branch occupies 1 byte only, and the "offset" is in the range 0 to 63, given in the bottom 6 bits. If bit 6 is clear, then the offset is a signed 14-bit number given in bits 0 to 5 of the first byte followed by all 8 of the second.

4.7.1

An offset of 0 means "return false from the current routine", and 1 means "return true from the current routine".

4.7.2

Otherwise, a branch moves execution to the instruction at address

```
  Address after branch data + Offset - 2.
```

4.8

Two opcodes, **print** and **print_ret**, are followed by a text string. This is stored according to the usual rules: in particular execution continues after the last 2-byte word of text (the one with top bit set).

Remarks

Some opcodes have type VAR only because the available codes for the other types had run out; **print_char**, for instance. Others, especially **call**, need the flexibility to have between 1 and 4 operands.

The Inform assembler can assemble branches in either form, though the programmer should always use long form unless there's a good reason. Inform automatically optimises branch statements so as to force as many of them as possible into short form. (This optimisation will happen to branches written by hand in assembler as well as to branches compiled by Inform.)

The disassembler **Txd** numbers locals from 0 to 14 and globals from 0 to 239 in its output (corresponding to variable numbers 1 to 15, and 16 to 255, respectively).

The branch formula is sensible because in the natural implementation, the program counter is at the address after the branch data when the branch takes place: thus it can be regarded as

```
  PC = PC + Offset - 2.
```

If the rule were simply "add the offset" then, since the offset couldn't be 0 or 1 (because of the return-false and return-true values), we would never be able to skip past a 1-byte instruction (say, a 0OP like **quit**), or specify the branch "don't branch at all" (sometimes useful to ignore the result of the test altogether). Subtracting 2 means that the only effects we can't achieve are

```
  PC = PC - 1     and     PC = PC - 2
```

and we would never want these anyway, since they would put the program counter somewhere back inside the same instruction, with horrid consequences.

On disassembly

Briefly, the first byte of an instruction can be decoded using the following table:

```
  $00 -- $1f  long      2OP     small constant, small constant
  $20 -- $3f  long      2OP     small constant, variable
  $40 -- $5f  long      2OP     variable, small constant
  $60 -- $7f  long      2OP     variable, variable
  $80 -- $8f  short     1OP     large constant
  $90 -- $9f  short     1OP     small constant
  $a0 -- $af  short     1OP     variable
  $b0 -- $bf  short     0OP
  except $be  extended opcode given in next byte
  $c0 -- $df  variable  2OP     (operand types in next byte)
  $e0 -- $ff  variable  VAR     (operand types in next byte(s))
```

Here is an example disassembly:

```
  @inc_chk c 0 label;     05 02 00 d4
      long form; count 2OP; opcode number 5; operands:
          02     small constant (referring to variable c)
          00     small constant 0
      branch if true: 1-byte offset, 20 (since label is
      18 bytes forward from here).
  @print "Hello.^";       b2 11 aa 46 34 16 45 9c a5
      short form; count 0OP.
      literal string, Z-chars: 4 13 10  17 17 20  5 18 5  7 5 5.
  @mul 1000 c -> sp;      d6 2f 03 e8 02 00
      variable form; count 2OP; opcode number 22; operands:
          03 e8  long constant (1000 decimal)
          02     variable c
      store result to stack pointer (var number 00).
  @call_1n Message;       8f 01 56
      short form; count 1OP; opcode number 15; operand:
          01 56  long constant (packed address of routine)
  .label;
```

---

5. How routines are encoded

5.1

A routine is required to begin at an address in memory which can be represented by a packed address (for instance, in Version 5 it must occur at a byte address which is divisible by 4).

5.2

A routine begins with one byte indicating the number of local variables it has (between 0 and 15 inclusive).

5.2.1

In Versions 1 to 4, that number of 2-byte words follows, giving initial values for these local variables. In Versions 5 and later, the initial values are all zero.

5.3

Execution of instructions begins from the byte after this header information. There is no formal 'end-marker' for a routine (it is simply assumed that execution eventually results in a return taking place).

5.4

In Version 6, there is a "main" routine (whose packed address is stored in the word at **$06** in the header) called when the game starts up. It is illegal to return from this routine.

5.5

In all other Versions, the word at **$06** contains the byte address of the first instruction to execute. The Z-machine starts in an environment with no local variables from which, again, a return is illegal.

Remarks

Note that it is permissible for a routine to be in dynamic memory. Marnix Klooster suggests this might be used for compiling code at run time!

In Versions 3 and 4, Inform always stores 0 as the initial values for local variables.

Inform's "main" routine is required not to have local variables and has to be the first defined routine. This ensures it is in the bottom 64K of memory, as it must be (in Versions other than 6).

---

6. The game state: storage and routine calls

6.1

The "state of play" is defined as the following: the contents of dynamic memory; the contents of the stack; the value of the program counter (PC), and the "routine call state" (that is, the chain of routines which have called each other in sequence, and the values of their local variables). Note that the routine call state, the stack and the PC must be stored outside the Z-machine memory map, in the interpreter's private memory.

6.1.1

The entire state of play must be stored when the game is saved.

6.1.1.1

The format of a saved game file is not specified.

6.1.1.2

An internal saved game for "undo" purposes (if there is one) is not part of the state of play. This is important: if a saved game file also contained the internal saved game at the time of saving, it would be impossible to undo the act of restoration. It also prevents internal saved games from growing larger and larger as they include their predecessors.

6.1.1.3

It is illegal to save the game (either with **save** or **save_undo**) during an "interrupt routine" (one coming about through timed input, sound effect termination or newline interrupts). Therefore saved games need not store information capable of restoring such a position.

6.1.2

On a "restore" or "undo" (which restores a game saved into internal memory), the entire state of play is written back except that 'Flags 2' in the header is preserved. (This information includes whether the game is being transcribed to printer and whether a fixed-pitch font is being used.)

6.1.2.1

Before a "restore", an interpreter should check that the file to be used has been saved from the same game currently being played. (See remark below.)

6.1.2.2

After a "restore" or "undo", an interpreter should reset the header values marked **Rst** in the header table of **S** 11. (It should not be assumed that the game was saved by the same interpreter.)

6.1.3

A "restart" is similar: the entire state is restored from the original story file, and the stack is emptied; but 'Flags 2' is preserved; and the interpreter should reset the **Rst** parts of the header.

6.1.4

In Versions 5 and later, an interpreter unable to save the game state into internal memory (for "undo" purposes) must clear bit 4 of 'Flags 2' in the header.

6.2

Global variables (variable numbers **$10** to **$ff**) are stored in a table in the Z-machine's dynamic memory, at a byte address given in word 6 of the header. The table consists of 240 2-byte words and the initial values of the global variables are the values initially contained in the table. (It is legal for a program to alter the table's contents directly in play, though not for it to change the table's address.)

6.3

Writing to the stack pointer (variable number **$00**) pushes a value onto the stack; reading from it pulls a value off. Stack entries are 2-byte words as usual.

6.3.1

The stack is considered as empty at the start of each routine: it is illegal to pull values from it unless values have first been pushed on.

6.3.2

The stack is left empty at the end of each routine: when a return occurs, any values pushed during the routine are thrown away.

6.3.3

Stack size has not previously been specified. The author proposes the present capacity of **Zip** as a future minimum standard: let the 'usage' of a routine call be 4 plus the number of local variables it has. During a game the total of the usages for each routine in the recursive chain of routines being called, plus the game's own stack usage, must never reach 1024.

6.4

Routine calls occur in the following circumstances: when one of the **call...** opcodes is executed; in Versions 4 and later, when timed keyboard input is being monitored; in Versions 5 and later, when a sound effect finishes; in Version 6, when the game begins (to call the "main" routine); in Version 6, when a "newline interrupt" occurs.

6.4.1

A routine call may have any number of arguments, from 0 to 3 (in Versions 1 to 4) or 0 to 7 (Versions 5 and later). All routines return a value (though sometimes this value is thrown away afterward: for example by opcodes in the form **call_vn***).

6.4.2

Routine calls preserve local variables and the stack (except when the return value is stored in a local variable or onto the top of the stack).

6.4.3

A routine call to packed address 0 is legal: it does nothing and returns false (0). Otherwise it is illegal to call a packed address where no routine is present.

6.4.4

When a routine is called, its local variables are created with initial values taken from the routine header (Versions 1 to 4) or with initial value 0 (Versions 5 and later). Next, the arguments are written into the local variables (argument 1 into local 1 and so on).

6.4.4.1

It is legal for there to be more arguments than local variables (any spare arguments are thrown away) or for there to be fewer.

6.4.5

The return value of a routine can be any Z-machine number. Returning 'false' means returning 0; returning 'true' means returning 1.

6.5

A "stack frame" is an index to the routine call state (that is, the call-stack of return addresses from routines currently running, and values of local variables within them). This index is a Z-machine number. The interpreter must be able to produce the current value and to set a value further down the call-stack than the current one, effectively throwing away its recent history (see **catch** and **throw**).

6.6

In Version 6, the Z-machine understands a third kind of stack: a "user stack", which is a table of words in dynamic memory. The first word in this table always holds the number of spare slots on the stack (so the initial value is the capacity of the stack). The Z-machine makes no check on stack under-flow (i.e., pulling more values than were pushed) which would over-run the length of the table if the program allowed it to happen.

Remarks

Some interpreters store the whole of dynamic memory to disc as part of their saved game files, which can make them as much as 45K or so long. A player making a serious attack on a game may end up wasting a whole megabyte, more than convenient without a hard disc. A technique invented by Bryan Scattergood, taken up by most modern interpreters, greatly reduces file size by only saving bytes of dynamic memory which differ from the initial state of the game.

It is unspecified how an interpreter should decide whether a saved game file belongs to the game currently being played. It is normal to insist that the release numbers, serial codes and checksums all match. The **Pinfocom** interpreter deliberately checks only the release number, so that saved games can be exchanged between different editions of 'Seastalker' (presumably compiled to handle the sonarscope differently).

These issues are taken up in great detail in Martin Frost's **Quetzal** standard for saved game files, created to allow different interpreters to exchange saved games. This Standard doesn't require compliance with **Quetzal**, but interpreter writers are urged to consider it: it can only help authors if players can send them saved games where bugs seem to have appeared.

The stack is stored in the interpreter's own memory, not anywhere in the Z-machine. The game program has no direct access to the stack memory or stack pointer; on some implementations the game's main stack is also used to store the routine call state (i.e. the game stack and the call-stack are the same) but this need not be true.

The stack size specification guarantees in particular that if the game itself never uses more than 32 stack entries at once then it can have a recursive depth of at least 90 routine calls. The author believes that old Infocom games will all run with a stack size of 512 words.

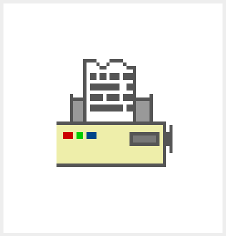
Note that the "state of play" does not include numerous input/output settings (the current window, cursor position, splitness or otherwise, which streams are selected, etc.): neither does it include the state of the random-number generator. (Games with elaborate status lines must redraw them after a restore has taken place.)

Zip provides "undo" but most versions of the **ITF** interpreter do not (and **save_undo** returns 0, unfortunately). This is probably its greatest failing. Some Infocom-written interpreters will only provide "undo" to a game which has bit 4 of 'Flags 2' set: but Inform 5.5 doesn't set this bit, so modern interpreters should be more generous.

Contents / Preface / Overview

Section 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9 / 10 / 11 / 12 / 13 / 14 / 15 / 16

Appendix A / B / C / D / E / F



7. Output streams and file handling

7.1

At any given time text is being output through a selection of "output streams" (possibly none, possibly several at once).

7.1.1

Two output streams are common to all Versions: number 1 (the screen) and 2 (the game transcript, usually printed to a printer or a file).

7.1.1.1

In Versions 1 to 5, the player's input to the **read** opcode should be echoed to output streams 1 and 2 (if stream 2 is active), so that text typed in appears in any transcript. In Version 6 input should be sent only to stream 1 and it is the game's responsibility to write to the transcript.

7.1.1.2

In Infocom's Version 4 game 'A Mind Forever Voyaging', which anticipated a printer rather than a file to receive the transcript, stream 2 is turned off and on again several times in quick succession. Thus if an interpreter decides where to send the transcript by asking the player for a filename, this question should only be asked once per game session, not every time stream 2 is selected.

7.1.2

Versions 3 and later supply these and two other output streams, numbered 3 (Z-machine memory) and 4 (a script file of the player's whole commands and of individual keypresses as read by **read_char**).

7.1.2.1

Output stream 3 writes to a table in dynamic memory. When the stream is selected, the table may have any contents (even the initial 'size' word will be ignored by the interpreter). While the stream is selected, the table's contents are unspecified (and a game cannot safely read or write to it). When the stream is deselected, the initial word of the table holds the number of characters printed and subsequent bytes hold those characters. Similarly, in Version 6, the total width of printing (in units) will then be stored in the word at **$30** in the header. (It is the programmer's responsibility to make the table large enough: the interpreter performs no overflow checking.)

7.1.2.1.1

******* It is possible for stream 3 to be selected while it is already on. If this happens, the previous table address is remembered and the previous table is resumed when the new one is finished. This nesting can reach a depth of up to 16: if stream 3 is opened for a seventeenth time, the interpreter should halt with an error message.

7.1.2.2

Output stream 3 is unusual in that, while it is selected, no text is sent to any other output streams which are selected. (However, they remain selected.)

7.1.2.2.1

Newlines are written to output stream 3 as ZSCII 13. (A game should never **print_char** the value 10, or any other value which is undefined as a ZSCII output code.)

7.1.2.3

Output stream 4 is unusual in that, when it is selected, the only text printed to it is that of the player's commands and keypresses (as read by **read_char**). (Each command is written, in one go, when it has been finished: a command which has been timed-out, or has been terminated by a code in the terminating character codes table, is not written. Mistypes and uses of 'delete' are not written.)

7.2

On output streams 1 and 2 (only), text printing may be "buffered" in that new-lines are automatically printed to ensure that no word (of length less than the width of the screen) spreads across two lines. (This process is sometimes called "word-wrapping".)

7.2.1

In Versions 1 to 3, buffering is always on. In Versions 4 and later it is on by default (at the start of a game) and a game can switch it on or off using the **buffer_mode** opcode.

7.2.2

In Version 6, each of the eight windows has its own "buffering flag". In Versions 3 to 5, the **buffer_mode** applies only to the lower window, and buffering never happens in the upper window.

7.3

In Versions 1 and 2, output stream 1 is always selected and stream 2 can be selected or deselected by the game, by setting or clearing bit 0 of 'Flags 2'.

7.4

In Versions 3 and later, all four output streams can be selected or deselected using the **output_stream** opcode. In addition, stream 2 can be selected or deselected by setting or clearing bit 0 of 'Flags 2'. Whichever method is used, the interpreter must ensure that this flag holds the current status of stream 2. ('A Mind Forever Voyaging' requires this.)

7.5

******* Because of the **print_unicode** opcode, it is possible for arbitrary Unicode characters to be sent to the output streams: that is, for characters which are not in the ZSCII set at all, even in the "extra characters" range.

7.5.1

See **S** 3.8.5.4 for rules on printing Unicode to stream 1.

7.5.2

Interpreters are free to use any representation of non-ASCII Unicode characters in stream 2. For example, they might print "[1a05]" to signify Unicode character **$1a05**; or they might be configurable to write transcript files which conform to any chosen ISO 8859 set.

7.5.3

When printed to stream 3, Unicode characters should be converted to ZSCII if possible. If this is not possible, a question mark should be printed to stream 3.

7.5.4

Non-ZSCII characters never need to be printed to stream 4.

7.6

******* In Versions 5 and later, the Z-machine has the ability to load and save files (using optional operands with the **save** and **restore** opcodes: these operands were not used in Infocom's Version 5 games, but I wish to specify them as in Version 5 anyway).

7.6.1

******* Filenames have the following format (approximately the MS-DOS 8.3 rule): one to eight alphanumeric characters, a full stop and zero to three alphanumeric characters (the "file extension").

7.6.1.1

The interpreter must convert all filenames to upper case before use. If no full stop is given, ".AUX" should be appended.

7.6.1.2

Games should avoid the extensions ".INF", ".H", ".Z" followed by a number or ".SAV": otherwise they may be in danger of erasing their own object code, source code or saved game files.

7.6.2

******* Saved files are not associated with any particular session of a game. They are not part of the "state of play".

7.6.3

******* A game may depend on having up to 32 auxiliary files (with different names).

7.6.4

File-handling errors such as "disc corrupt" and "disc full" should be reported directly to the player by the interpreter. The error "file not found" should only cause a failure return code from **restore**.

Remarks

The **ITF** interpreter incorrectly applies buffering when printing to the upper window.

Note that the requirement 7.1.2.1.1, that usages of stream 3 can be 'nested', is new in Standard 1.0. This is potentially important for Inform games, as stream 3 is often used to examine text before printing, for instance to choose between the articles "a" and "an" in front of an object name. But the process of printing an object name may itself require a usage of stream 3, and so on.

An ambiguous point about output stream 4 is whether it should contain the answers to interpreter questions like "what file name should your saved game have?": it can actually be quite useful to be able to include such answers in test script files. (When running a long script, I often save the game at several places during it, in order to save time in re-running passages.)

An interpreter should be able to write time delays (for timed input), accented characters or mouse clicks into stream 4 (i.e., to a script file). One possible style to record this information might be:

```
    take lamp              an ordinary command
    turn it on.[154]       command, full stop, then keypad 9
                           (which might abbreviate for NE)
    look unde[0]           timed out input
    look under the rock    the same input continuing
    [254][10][6]           mouse-click at (10,6)
```

A typical auxiliary file might be one containing the player's preferred choices. This would be created when he first changed any of the default settings, and loaded (if present) whenever the game started up.

---

8. The screen model

8.1

Text may be printed in any font of the interpreter's choice, variable- or fixed-pitch: except that when bit 1 of 'Flags 2' in the header is set, or when the text style has been set to Fixed Pitch, then a fixed-pitch font must be used.

8.1.1

In Version 5, the height and width of the current font (in units (see below)) should be written to bytes **$27** and **$26** of the header, respectively. In Version 6, these bytes are the other way round (height in **$27**, width in **$26**). The width of a font is defined as the width of its '0' character.

8.1.2

An interpreter should ideally provide 4 fonts, with ID numbers as follows:

```
   1: the normal font
   2: a picture font
   3: a character graphics font
   4: a Courier-style font with fixed pitch
```

(In addition, font ID 0 means "the previous font".) Ideally all text styles should be available for each font (for instance, Courier bold should be obtainable) except that font 3 need only be available in Roman and Reverse Video. Each font should provide characters for character codes 32 to 126 (plus character codes for any accented characters with codes greater than 127 which are being implemented as single accented letters on-screen).

8.1.3

******* A game must not use fonts other than 1 unless allowed to by the interpreter: see the **set_font** opcode for how to give or refuse permission. (This paragraph is marked ******* because existing Infocom games determined the availability of font 3 for 'Beyond Zork' in a complicated and unsatisfactory way: see **S** 16.)

8.1.3.1

***** It is legal for a game to change font at any time, including halfway through the printing of a word. (This might be needed to introduce exotic foreign accents in the future.)

8.1.4

The specification of the "picture font" is unknown (conjecturally, it was intended to provide pictures before Version 6 was properly developed). Interpreters need not implement it.

8.1.5

The specification of the character graphics font is given in **S** 16.

8.1.5.1

In Version 5 (only), an interpreter which cannot provide the character graphics font should clear bit 3 of 'Flags 2' in the header.

8.2

In Versions 1 to 3, a status line should be printed by the interpreter, as follows. In Version 3, it must set bit 4 of 'Flags 1' in the header if it is unable to produce a status line.

8.2.1

In Versions 1 and 2, all games are "score games". In Version 3, if bit 1 of 'Flags 1' is clear then the game is a "score game"; if it is set, then the game is a "time game".

8.2.2

The short name of the object whose number is in the first global variable should be printed on the left hand side of the line.

8.2.2.1

Whenever the status line is being printed the first global must contain a valid object number. (It would be useful if interpreters could protect themselves in case the game accidentally violates this requirement.)

8.2.2.2

If the object's short name exceeds the available room on the status line, the author suggests that an interpreter should break it at the last space and append an ellipsis "...". There is no guaranteed maximum length for location names but an interpreter should expect names of length up to at least 49 characters.

8.2.3

If there is room, the right hand side of the status line should display:

8.2.3.1

For "score games": the score and number of turns, held in the values of the second and third global variables respectively. The score may be assumed to be in the range -99 to 999 inclusive, and the turn number in the range 0 to 9999.

8.2.3.2

For "time games": the time, in the form **hours:minutes** (held in the second and third globals). The time may be given on a 24-hour clock or the number of hours may be reduced modulo 12 (but if so, "AM" or "PM" should be appended). Either way the player should be able to see the difference between 4am and 4pm, for example. The hours global may be assumed to be in the range 0 to 23 and the minutes global in the range 0 to 59.

8.2.4

The status line is updated in exactly two circumstances: when a **show_status** opcode is executed, and just before the keyboard is read by **read**. (It is not displayed when the game begins.)

8.3

Under Versions 5 and later, text printing has a current foreground and background colour. In Version 6, each window has its own pair. (Note that a Version 6 interpreter going under the Amiga interpreter number must use the same pair of colours for all windows. If either is changed, then the interpreter must change the colour of all text on the screen to match. This simulates the Amiga hardware, which used two logical colours for text and switched palette to change their physical colour.)

8.3.1

The following codes are used to refer to colours:

```
    -1 =  the colour of the pixel under the cursor (if any)
    0  =  the current setting of this colour
    1  =  the default setting of this colour
    2  =  black   3 = red       4 = green    5 = yellow
    6  =  blue    7 = magenta   8 = cyan     9 = white
    10 =  darkish grey (MSDOS interpreter number)
    10 =  light grey   (Amiga interpreter number)
    11 =  medium grey  (ditto)
    12 =  dark grey    (ditto)
```

Colours 10, 11, 12 and -1 are available only in Version 6. In Version 6 the pictures in some graphics files use colours beyond the above: if so the result of "the colour under the cursor" is permitted to be stored with value 16 or greater.

8.3.2

If the interpreter cannot produce colours, it should clear bit 0 of 'Flags 1' in the header. In Version 6 it should write colours 2 and 9 (black and white), either way round, into the default background and foreground colours in bytes **$2c** and **$2d** of the header.

8.3.3

If the interpreter can produce colours, it should set bit 0 of 'Flags 1' in the header, and write its default background and foreground colours into bytes **$2c** and **$2d** of the header.

8.3.4

If a game wishes to use colours, it should have bit 6 in 'Flags 2' set in its story file. (However, an interpreter should not rule out the use of colours just because this has not been done.)

8.4

The screen should ideally be at least 60 characters wide by 14 lines deep. (Old Apple II interpreters had a 40 character width and some modern laptop ones have a 9 line height, but implementors should seek to avoid these extremes if possible.) The interpreter may change the exact dimensions whenever it likes but must write the current height (in lines) and width (in characters) into bytes **$20** and **$21** in the header.

8.4.1

The interpreter should use the screen height for calculating when to pause and print "[MORE]". A screen height of 255 lines means "infinite height", in which case the interpreter should never stop printing for a "[MORE]" prompt. (In case, say, the screen is actually a teletype printer, or has very good "scrollback".)

8.4.2

Screen dimensions are measured in notional "units". In Versions 1 to 4, one unit is simply the height or width of one character. In Version 5 and later, the interpreter is free to implement units as anything from character sizes down to individual pixels.

8.4.3

In Version 5 and later, the screen's width and height in units should be written to the words at **$22** and **$24**.

8.5

The screen model for Versions 1 and 2 is as follows:

8.5.1

The screen can only be printed to (like a teletype) and there is no control of the cursor.

8.5.2

At the start of a game, the screen should be cleared and the text cursor placed at the bottom left (so that text scrolls upwards as the game gets under way).

8.6

The screen model for Version 3 is as follows:

8.6.1

The screen is divided into a lower and an upper window and at any given time one of these is selected. (Initially it is the lower window.) The game uses the **set_window** opcode to select one of the two. Each window has its own cursor position at which text is printed. Operations in the upper window do not move the cursor of the lower. Whenever the upper window is selected, its cursor position is reset to the top left. Selecting, or re-sizing, the upper window does not change the screen's appearance.

8.6.1.1

The upper window has variable height (of n lines) and the same width as the screen. This should be displayed on the n lines of the screen below the top one (which continues to hold the status line). Initially the upper window has height 0. When the lower window is selected, the game can split off an upper window of any chosen size by using the **split_window** opcode.

8.6.1.1.1

Printing onto the upper window overlays whatever text is already there.

8.6.1.1.2

When a screen split takes place in Version 3, the upper window is cleared.

8.6.1.2

An interpreter need not provide the upper window at all. If it is going to do so, it should set bit 5 of 'Flags 1' in the header to signal this to the game. It is only legal for a game to use **set_window** or **split_window** if this bit has been set.

8.6.1.3

Following a "restore" of the game, the interpreter should automatically collapse the upper window to size 0.

8.6.2

When text reaches the bottom right of the lower window, it should be scrolled upwards. The upper window should never be scrolled: it is legal for a character to be printed on the bottom right position of the upper window (but the position of the cursor after this operation is undefined: the author suggests that it stay put).

8.6.3

At the start of a game, the screen should be cleared and the text cursor placed at the bottom left (so that text scrolls upwards as the game gets under way).

8.7

The screen model for Versions 4 and later, except Version 6, is as follows:

8.7.1

Text can be printed in five different styles (modelled on the VT100 design of terminal). These are: Roman (the default), Bold, Italic, Reverse Video (usually printed with foreground and background colours reversed) and Fixed Pitch. The specification does not require the interpreter to be able to display more than one of these at once (e.g. to combine italic and bold), and most interpreters can't. If the interpreter is going to allow certain combinations, then note that changing back to Roman should turn off all the text styles currently active.

8.7.1.1

An interpreter need not provide Bold or Italic (even for font 1) and is free to interpret them broadly. (For example, rendering bold-face by changing the colour, or rendering italic with underlining.)

8.7.1.2

It is legal to change text style at any point, including in the middle of a word being printed.

8.7.2

There are two "windows", called "upper" and "lower": at any given time one of these two is selected. (Initially it is the lower window.) The game uses the **set_window** opcode to select one of the two. Each window has its own cursor position at which text is printed. Operations in the upper window do not move the cursor of the lower. Whenever the upper window is selected, its cursor position is reset to the top left.

8.7.2.1

The upper window has variable height (of n lines) and the same width as the screen. (It is usual for interpreters to print the upper window on the top n lines of the screen, overlaying any text which is already there, having been printed in the lower window some time ago.) Initially the upper window has height 0. When the lower window is selected, the game can split off an upper window of any chosen size by using the **split_window** opcode.

8.7.2.1.1

It is unclear exactly what **split_window** should do if the upper window is currently selected. The author suggests that it should work as usual, leaving the cursor where it is if the cursor is still inside the new upper window, and otherwise moving the cursor back to the top left. (This is analogous to the Version 6 practice.)

8.7.2.2

In Version 4, the lower window's cursor is always on the bottom screen line. In Version 5 it can be at any line which is not underneath the upper window. If a split takes place which would cause the upper window to swallow the lower window's cursor position, the interpreter should move the lower window's cursor down to the line just below the upper window's new size.

8.7.2.3

When the upper window is selected, its cursor position can be moved with **set_cursor**. This position is given in characters in the form (row, column), with (1,1) at the top left. The opcode has no effect when the lower window is selected. It is illegal to move the cursor outside the current size of the upper window.

8.7.2.4

An interpreter should use a fixed-pitch font when printing on the upper window.

8.7.2.5

In Versions 3 to 5, text buffering is never active in the upper window (even if a game begins printing there without having turned it off).

8.7.3

Clearing regions of the screen:

8.7.3.1

When text reaches the bottom right of the lower window, it should be scrolled upwards. (When the text style is Reverse Video the new blank line should **not** have reversed colours.) The upper window should never be scrolled: it is legal for a character to be printed on the bottom right position of the upper window (but the position of the cursor after this operation is undefined: the author suggests that it stay put).

8.7.3.2

Using the opcode **erase_window**, the specified window can be cleared to background colour. (Even if the text style is Reverse Video the new blank space should not have reversed colours.)

8.7.3.2.1

In Versions 5 and later, the cursor for the window being erased should be moved to the top left. In Version 4, the lower window's cursor moves to its bottom left, while the upper window's cursor moves to top left.

8.7.3.3

Erasing window -1 clears the whole screen to the background colour of the lower screen, collapses the upper window to height 0, moves the cursor of the lower screen to bottom left (in Version 4) or top left (in Versions 5 and later) and selects the lower screen. The same operation should happen at the start of a game.

8.7.3.4

Using **erase_line** in the upper window should erase the current line from the cursor position to the right-hand edge, clearing it to background colour. (Even if the text style is Reverse Video the new blank space should not have reversed colours.)

8.8

The screen model for Version 6 is as follows:

8.8.1

The display is an array of pixels. Coordinates are usually given (in units) in the form (y,x), with (1,1) in the top left.

8.8.2

If the interpreter thinks the status line should be redrawn (e.g. because a menu window has been clicked over it), it may set bit 2 of 'Flags 2'. The game is expected to notice, take action and clear the bit. (However, a more efficient interpreter would cache the status line and handle redraws itself.)

8.8.3

There are eight "windows", numbered 0 to 7. The code -3 is used as a window number to mean "the currently selected window". This selection can be changed with the **set_window** opcode. Windows are invisible and usually lie on top of each other. All text and graphics plotting is always clipped to the current window, and anything showing through is plotted onto the screen. Subsequent movements of the window do not move what was printed and there is no sense in which characters or graphics 'belong' to any particular window once printed. Each window has a position (in units), a size (in units), a cursor position within it (in units, relative to its own origin), a number of flags called "attributes" and a number of variables called "properties".

8.8.3.1

There are four attributes, numbered as follows:

```
    0: wrapping
    1: scrolling
    2: text copied to output stream 2 (the transcript, if selected)
    3: buffered printing
```

Each can be turned on or off, using the **window_style** opcode.

8.8.3.1.1

"Wrapping" is the continuation of printed text from one line to the next. Text running up to the right margin will continue from the left margin of the following line. If "wrapping" is off then characters will be printed until no more can be fitted in without hitting the right margin, at which point the cursor will move to the right margin and stay there, so that any further text will be ignored.

8.8.3.1.2

"Buffered printing" means that text to be printed in the window is temporarily stored in a buffer and only flushed onto the screen at intervals convenient for the interpreter.

8.8.3.1.2.1

"Buffered printing" has two practical effects: firstly it causes a delay before printed text actually appears.

8.8.3.1.2.2

Secondly it affects the way "wrapping" is done. If "buffered printing" is on, then text is wrapped after the last word which could fit on a line. If not, then text is wrapped after the last character that could fit.

Example: suppose the text "Here is an abacus" is printed in a narrow window. The appearance (after the buffer has been flushed, if there is buffered printing) might be:

```
                                 |...margins....|
    wrapping on    buffering on        Here is an
                                       abacus^
             off   buffering on        Here is an aba^

    wrapping on    buffering off       Here is an aba
                                       cus^
             off   buffering off       Here is an aba^
```

where the caret denotes the final position of the cursor. (Games often alter "wrapping": it would normally be on for a window holding running text but off for a status-line window, which is why window 0 has "wrapping" on by default but all other windows have "wrapping" off by default. On the other hand all windows have "buffered printing" on by default and games only alter this in rare circumstances to avoid delays in the appearance of individual printed characters.)

8.8.3.2

There are 16 properties, numbered as follows:

```
    0  y coordinate    6   left margin size            12  font number
    1  x coordinate    7   right margin size           13  font size
    2  y size          8   newline interrupt routine   14  attributes
    3  x size          9   interrupt countdown         15  line count
    4  y cursor        10  text style
    5  x cursor        11  colour data
```

Each property is a standard Z-machine number and is readable with **get_wind_prop** and writeable with **put_wind_prop**. However, a game should only use **put_wind_prop** to set the newline interrupt routine, the interrupt countdown and the line count: everything else is either set by the interpreter or by specialised opcodes (such as **set_font**).

8.8.3.2.1

If a window has character wrapping, then text is clipped to stay inside the left and right margins. After a new-line, the cursor moves to the left margin on the next line. Margins can be set with **set_margins** but this should only be done just after a newline or just after the window has been selected. (These values are margin sizes in pixels, and are by default 0.)

8.8.3.2.2

If the interrupt countdown is set to a non-zero value (which by default it is not), then the line count is decremented on each new-line, and when it hits zero the routine whose packed address is stored in the "newline interrupt routine" property is called before text printing resumes. (This routine may, for example, meddle with margins to roll text around a crinkly-shaped picture.) The interrupt routine should not attempt to print anything.

8.8.3.2.2.1

Because of an Infocom bug, if the interpreter number is 6 (for MSDOS) and the story file is 'Zork Zero' release 393.890714, but in no other case, the interpreter must do the following instead: (1) move to the new line, (2) put the cursor at the current left margin, (3) call the interrupt routine (if it's time to do so). This is the least bad way to get around a basic inconsistency in existing Infocom story files and interpreters.

8.8.3.2.2.2

Note that the **set_margins** opcode, which is often used by newline interrupt routines (to adjust the shape of a margin as it flows past a picture), automatically moves the cursor if the change in margins would leave the cursor outside them. The effect will depend, unfortunately, on which sequence of events above takes place.

8.8.3.2.2.3

A line count is never decremented below -999.

8.8.3.2.3

The text style is set just as in Version 4, using **set_text_style** (which sets that for the current window). The property holds the operand of that instruction (e.g. 4 for italic).

8.8.3.2.4

The foreground colour is stored in the lower byte of the colour data property, the background colour in the upper byte.

8.8.3.2.5

The font height (in pixels) is stored in the upper byte of the font size property, the font width (in pixels) in the lower byte.

8.8.3.2.6

The interpreter should use the line count to see when it should print "[MORE]". A line count of -999 means "never print [MORE]". (Version 6 games often set line counts to manipulate when "[MORE]" is printed.)

8.8.3.2.7

If an attempt is made by the game to read the cursor position at a time when text is held unprinted in a buffer, then this text should be flushed first, to ensure that the cursor position is accurate before being read.

8.8.3.3

All eight windows begin at (1,1). Window 0 occupies the whole screen and is initially selected. Window 1 is as wide as the screen but has zero height. Windows 2 to 7 have zero width and height. Window 0 initially has attribute 1 off and 2, 3 and 4 on (scrolling, copy to printer transcript, buffering). Windows 1 to 7 initially have attribute 4 (buffering) on, and the other attributes off.

8.8.3.4

A window can be moved with **move_window** and resized with **window_size**. If the window size is reduced so that its cursor lies outside it, the cursor should be reset to the left margin on the top line.

8.8.3.5

Each window remembers its own cursor position (relative to its own coordinates, so that the position (1,1) is at its top left). These can be changed using **set_cursor** (and it is legal to move the cursor for an unselected window). It is illegal to move the cursor outside the current window.

8.8.3.6

Each window can be scrolled vertically (up or down) any number of pixels, using the **scroll_window** opcode.

8.8.4

To some extent windows 0 and 1 mimic the behaviour of the lower and upper windows in the Version 4 screen model:

8.8.4.1

The **split_screen** opcode tiles windows 0 and 1 together to fill the screen, so that window 1 has the given height and is placed at the top left, while window 0 is placed just below it (with its height suitably shortened, possibly making it disappear altogether if window 1 occupies the whole screen).

8.8.4.2

An "unsplit" (that is, a **split_screen 0**) takes place when the entire screen is cleared with **erase_window -1**, if a "split" has previously occurred (meaning that windows 0 and 1 have been set up as above).

8.8.5

Screen clearing operations:

8.8.5.1

Erasing a picture is like drawing it (see below), except that the space where it would appear is painted over with background colour instead.

8.8.5.2

The current line can be erased using **erase_line**, either all the way to the right margin or by any positive number of pixels in that direction. The space is painted over with background colour (even if the current text style is Reverse Video).

8.8.5.3

Each window can be erased using **erase_window**, erasing to background colour (even if the current text style is Reverse Video).

8.8.5.3.1

Erasing window number -1 erases the entire screen to the background colour of window 0, unsplits windows 0 and 1 (see **S** 8.7.3.3 above) and selects window 0.

8.8.5.3.2

Erasing window -2 erases the entire screen to the current background colour. (It doesn't perform **erase_window** for all the individual windows, and it doesn't change any window attributes or cursor positions.)

8.8.6

Pictures may accompany the game. They are not stored in the story file (or the Z-machine) itself, and the interpreter is simply expected to know where to find them.

8.8.6.1

Pictures are numbered from 1 upwards (not necessarily contiguously). They can be "drawn" or "erased" (using **draw_picture** and **erase_picture**). Before attempting to do so, a game may ask the interpreter about the picture (using **picture_data**): this allows the interpreter to signal that the picture in question is unavailable, or to specify its height and width.

8.8.6.2

The game may, if it wishes, use the **picture_table** opcode to give the interpreter advance warning that a group of pictures will soon be needed (for instance, a collection of icons making up a control panel). The interpreter may want to load these pictures off disc and into a memory cache.

Remarks

See **S** 16 for comment on how 'Beyond Zork' uses fonts.

Some interpreters print the status line when they begin running a Version 3 game, but this is incorrect. (It means that a small game printing text and then quitting cannot be run unless it includes an object.) The author's preferred status line formats are:

```
Hall of Mists                                  80/733
Lincoln Memorial                               12:03 PM
```

Thus the score/turns block always fits in 3+1+4=8 characters and the time in 2+1+2+1+2=8 characters. (Games needing more exotic time lines, for example, should not be written in Version 3.)

The only existing Version 3 game to use an upper window is 'Seastalker' (for its sonarscope display).

Some ports of **ITF** apply buffering (i.e. word-wrapping) and scrolling to the upper window, with unfortunate consequences. This is why the standard Inform status line is one character short of the width of the screen.

The original Infocom files seldom use **erase_window**, except with window -1 (for instance 'Trinity' only uses it in this form). **ITF** does not implement it in any other case.

The Version 5 re-releases of older games make use of consecutive **set_text_style** instructions to attempt to combine boldface reverse video (in the hints system).

None of Infocom's Version 4 or 5 files use **erase_line** at all, and **ITF** implements it badly (with unpredictable behaviour in Reverse Video text style). (It's interesting to note that the Version 5 edition of 'Zork I' - one of the earliest Version 5 files -- blanks out lines by looking up the screen width and printing that many spaces.)

It's recommended that a Version 5 interpreter always use units to correspond to characters: that is, characters occupy 1×1 units. 'Beyond Zork' was written in the expectation that it could be using either 1x1 or 8x8, and contains correct code to calculate screen positions whatever units are used. (Infocom's Version 5 interpreter for MSDOS could either run in a text mode, 1x1, or a graphics mode, 8x8.) However, the German translation of 'Zork I' contains incorrect code to calculate screen positions unless 1x1 units are used.

Note that a minor bug in **Zip** writes bytes **$22** to **$25** in the header as four values, giving the screen dimensions in the form left, right, top, bottom: provided units are characters (i.e. provided the font width and height are both 1) then since "left" and "top" are both 0, this bug has no effect.

Some details of the known IBM graphics files are given in Paul David Doherty's "Infocom Fact Sheet". See also Mark Howell's program "pix2gif", which extracts pictures to GIF files. (This is one of his "Ztools" programs.)

Although Version 6 graphics files are not specified here, and were released in several different formats by Infocom for different computers, a consensus seems to have emerged that the MCGA pictures are the ones to adopt (files with filenames ***.MG1**). These are visually identical to Amiga pictures (whose format has been deciphered by Mark Knibbs). However, some Version 6 story files were tailored to the interpreters they would run on, and use the pictures differently according to what they expect the pictures to be. (For instance, an Amiga-intended story file will use one big Amiga-format picture where an MSDOS-intended story file will use several smaller MCGA ones.)

The easiest option is to interpret only DOS-intended Version 6 story files and only MCGA pictures. But it may be helpful to examine the **Frotz** source code, as **Frotz** implements **draw_picture** and **picture_data** so that Amiga and Macintosh forms of Version 6 story files can also be used.

It is generally felt that newly-written graphical games should not imitate the old Infocom graphics formats, which are very awkward to construct and have been overtaken by technology. Instead, the draft **Blorb** proposal for packaging up resources with Z-machine games calls for PNG format graphics glued together in a fairly simple way. An ideal Version 6 interpreter ought to understand *both* the four Infocom picture-sets *and* any **Blorb** set, thus catering for old and new games alike.

The line count of -999 preventing "[MORE]" is a device used by the demonstration mode of 'Zork Zero'.

Infocom's Version 6 interpreters and story files disagree on the meaning of window attributes 0 and 3 and the opcode **buffer_mode**, in such a way that the original specification is hard to deduce from the final behaviour. If we call the three possible ways that text can appear "word wrap", "char wrap" and "char clip":

```
                   |...margins....|
    word wrap       Here is an
                    abacus^
    char wrap       Here is an aba
                    cus^
    char clip       Here is an aba^
```

then Infocom's interpreters behave as follows:

```
                  Apple II      MSDOS         Macintosh   Amiga
```

A0 off, A3 off	char clip(LR)	char clip()	---	---
A0 off, A3 on	char clip(LR)	char clip(LR)	---	---
A0 on, A3 off	word wrap	char wrap	---	---
A0 on, A3 on	word wrap	word wrap	---	---
buffer_mode off	---	---	char wrap	char clip(L)
buffer_mode on	---	---	word wrap	word wrap

Here "---" means that the interpreter ignores the given state, and the presence of L, R or both after "char clipp" indicates which of the left and right margins are respected. The Amiga behaviour may be due to a bug and two bugs have also been found in the MSDOS implementation. Under this standard, the appearance is as follows:

	Standard
A0 off, A3 off	char clip(LR)
A0 off, A3 on	char clip(LR)
A0 on, A3 off	char wrap
A0 on, A3 on	word wrap
buffer_mode off	---
buffer_mode on	---

Due to a bug or an oversight, the V6 story files for all interpreters use **buffer_mode** once: to remove buffering while printing "Please wait..." with a row of full stops trickling out during a slow operation. Buffering would frustrate this, but fortunately on modern computers the operation is no longer slow and so the bug does not cause trouble.

Contents / Preface / Overview

Section 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9 / 10 / 11 / 12 / 13 / 14 / 15 / 16

Appendix A / B / C / D / E / F

---

9. Sound effects

9.1

Some games, from Version 3 onward, have sound effects attached. These are not stored in the story files (or the Z-machine) itself, and the interpreter is simply expected to know where to find them. Other games have only one sound effect, usable in a much more restricted way: a beep or bell sound, which we shall call a "bleep".

9.1.1

In Version 6, the interpreter should set bit 5 of 'Flags 1' if it can provide sound effects beyond a bleep.

9.1.2

In Version 5 and later, a game should have bit 7 of 'Flags 2' set in its story file if it wants to use sound effects beyond a bleep. The interpreter should then clear this bit if it cannot oblige.

9.2

Sound effects are numbered upwards from 1. Number 1 is a high-pitched bleep, number 2 a low-pitched one and effects from 3 upward are supplied by the interpreter somehow for the particular game in question.

9.3

Sound effects (other than bleeps) can be played at any volume level from 1 to 8 (8 being loudest of these). The volume level -1 should be implemented as "loudest possible".

9.4

Bleeps are immediate and brief. Other sound effects take place in the background, while normal operation of the Z-machine is going on. Control is via the **sound_effect** opcode, allowing the game to prepare, start, stop or finish with an effect.

9.4.1

The game may (but need not) "prepare" a sound effect before use. This would indicate to the interpreter that the game intends to use the effect soon: an interpreter might act on this information by loading the sampled sound off disc and into a memory cache.

9.4.2

A sound effect (other than a bleep) can then be "stopped" or "started". Only one sound effect is playing at any given time, and starting a new sound effect automatically stops any current one.

9.4.3

In Versions 5 and later, a sound effect may repeat any specified number of times, or repeat forever (until stopped).

9.4.4

Eventually, though, if it has not been stopped, it may end by itself. A routine (specified at start time) can then be called. The intention is that this routine may implement effects such as fading in and out, by replaying the sound effect at a different volume. (A game should not place any important code in such a routine.)

9.4.5

The game may, but need not, explicitly "finish with" any sound effect which is not likely to occur again for a while: the interpreter can then throw it out of memory.

Remarks

The safest way an Inform program can try to produce a bleep is by executing **@sound_effect 1**. Some ports of **Zip** believe that the first operand of this is the number of bleeps to make (so that **@sound_effect 2** bleeps twice), but this is incorrect.

Several Infocom games bleep (using **sound_effect** with only one operand, always equal to 1 or 2). Two provided sampled sound effects but did not bleep: 'The Lurking Horror' and 'Sherlock'. Their story files contain the following usages of **sound_effect**:

```
  sound_effect number 2 volume                   (in TLH)
  sound_effect number 2 volume/repeats function  (in Sherlock)
  sound_effect 0 3
  sound_effect number 3
  sound_effect 0 4
```

except that, probably due to a bug in its own code, 'TLH' can also generate

```
  sound_effect 4 8
  sound_effect 4095 2 15
```

A further difficulty with 'TLH' is that it assumes the interpreter is as slow as Infocom's Amiga interpreter was: it fires off several sound effects in one game round, assuming there will be time for it to play most of each one. To simulate this, **sound_effect** must be rewritten to pause sometimes:

if a new sound effect is begun while there is still one playing which was started since the last keyboard input, then wait until that earlier one finishes one cycle before replacing it with the new sound effect.

Infocom's MS-DOS interpreters for V4 to V6 set bit 5 of 'Flags 1' in all circumstances (i.e., whether or not sound effects are available). This would be incorrect behaviour for a standard interpreter.

Infocom implemented sound effects differently on different machines. The format of Infocom's shipped sound effects files has been documented by Stefan Jokisch and his notes are available from **ftp.gmd.de**. See also Andrew Plotkin's draft **Blorb** format for a more modern way to make sound effects available to newer games.

Contents / Preface / Overview

Section 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9 / 10 / 11 / 12 / 13 / 14 / 15 / 16

Appendix A / B / C / D / E / F

---

10. Input streams and devices

10.1

In Versions 1 and 2, the player's commands can only be drawn from the keyboard.

10.2

In Versions 3 and later, the player's keypresses are drawn from the current "input stream". There are two input streams: numbered 0 (the keyboard) and 1 (a file containing commands). Other inputs (mouse clicks or menu selections), if available, are also implemented as keypresses (see below).

10.2.1

The format of a file containing commands must be the same as that written in output stream 4.

10.2.2

The game can change the current input stream itself, using the opcode **input_stream**. It has no way of finding out which input stream is currently in use. An interpreter is free to change the input stream whenever it likes (e.g. at the player's request) or, indeed, to run the entire game under input stream 1 (for testing purposes).

10.2.3

When input stream 1 is first selected, the interpreter may use any method of choosing a file name for the file of commands. (Good practice is to use the same conventions as when choosing a filename for output to stream 4.)

10.2.4

When the the current stream is stream 1, the interpreter should not hold up long passages of text (by printing "[MORE]" and waiting for a keypress, for instance).

10.3

Mouse support is optional but can be provided in Versions 5 and later.

10.3.1

In a game which wishes to use the mouse, bit 5 of 'Flags 2' in the header should be set in the story file. If it wishes to read the mouse position after clicks, it must provide at least the first two words of a header extension table. (Note that Inform 6.12 and later always provide a header extension table at least this large, but Inform 6.11 and earlier never provide an extension table at all.)

10.3.1.1

If the interpreter cannot offer mouse support, then it should clear bit 5 of 'Flags 2' to signal this to the game.

10.3.2

Whenever a mouse click takes place (and provided the header extension table exists and contains at least 2 words) the interpreter should update the coordinates as follows:

```
   Word 1:  x coordinate where click took place
   Word 2:  y coordinate where click took place
```

10.3.3

The mouse is presumed to have between 0 and 16 buttons. The state of these buttons can be read by the **read_mouse** opcode in Version 6. Otherwise, mouse clicks are treated as keyboard input codes (see below).

10.3.4

In Version 6, the mouse can either be free or constrained to one of the 8 windows: if so, clicks outside the 'mouse window' must be ignored, and the interpreter is at liberty to confine the mouse's movement to the boundary of its window.

10.4

Menu support can optionally be provided in Version 6.

10.4.1

In a game which wishes to use menus, bit 8 of 'Flags 2' in the header should be set in the story file.

10.4.1.1

If the interpreter cannot offer menu support, then it should clear bit 8 of 'Flags 2' to signal this to the game.

10.4.2

Menus are numbered from 0 upwards. 0, 1 and 2 are reserved for the interpreter to manage (this system has only been implemented on the Macintosh, wherein 0 is the Apple menu, 1 the File menu and 2 the Edit menu). Menus numbered 3 and upwards can be created or removed with the **make_menu** opcode.

10.4.3

Menu selection is reported to the game as a keypress (see below). Details of what selection has been made are read with **read_mouse**.

10.5

Whole commands are read from the input stream using the **read** opcode. (Note that this has two different internal names in Inform, **sread** for Versions 1 to 4 and **aread** subsequently.)

10.5.1

In Versions 1 to 3, the interpreter must redisplay the status line before it begins accepting input.

10.5.2

Commands are normally terminated by a new-line (a carriage return or a line feed as appropriate for the machine's keyboard or file format).

10.5.2.1

In Versions 5 and later, the game may provide a "terminating characters table" by giving its byte address in the word at **$2e** in the header. This table is a zero-terminated list of input character codes which cause **aread** to finish the command (in addition to new-line). Only function key codes are permitted: these are defined as those between 129 and 154 inclusive, together with 252, 253 and 254. The special value 255 means "any function key code is terminating".

10.5.3

******* In Versions 4 and later, an interpreter should ideally be able to time input and to call a (game) routine at periodic intervals: see the **read** opcode. If it is able to do this, it should set bit 7 of 'Flags 1' in the header.

10.6

In Versions 4 and later, individual characters can be read from the current input stream, using **read_char**. Again, the interpreter should ideally be able to time input and to call a (game) routine at periodic intervals. If it is able to do this, it should set bit 7 of 'Flags 1' in the header.

10.7

The only characters which can be read from the keyboard are ZSCII characters defined for input (see **S** 3).

10.7.1

Every ZSCII character defined for input can be returned by **read_char**.

10.7.2

Only ZSCII characters defined for both input and output can be stored in the text buffer supplied to the **read** opcode.

10.7.3

The "escape" code is optional: that is, an interpreter need not provide an escape key. (The Inform library clears and quits menus if this code is returned to **read_char**.)

Remarks

Menus in 'Beyond Zork' define cursor up and cursor down as terminating characters, and make use of **read** in the upper window.

Contents / Preface / Overview

Section 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9 / 10 / 11 / 12 / 13 / 14 / 15 / 16

Appendix A / B / C / D / E / F

---

11. The format of the header

11.1

The header table summarises those locations in the Z-machine's header which an interpreter must deal with. (For further notes on traditional usage, see Appendix B.) "Hex" means the address, in hexadecimal; "V" the earliest Version to which the rule is applicable; "Dyn" means that the byte or bit may legally be changed by the game during play; "Int" means that the interpreter may change it; "Rst" means that the interpreter must set it correctly after loading the game, after a restore or after a restart.

Header format

Hex	V	Dyn	Int	Rst	Contents
0	1				Version number (1 to 6)
1	3				*Flags 1 (in Versions 1 to 3):*
					Bit 1: Status line type: 0=score/turns, 1=hours:mins
					2: Story file split across two discs?
			*	*	4: Status line not available?
			*	*	5: Screen-splitting available?
			*	*	6: Is a variable-pitch font the default?
	4				*Flags 1 (from Version 4):*
	5		*	*	Bit 0: Colours available?
	6		*	*	1: Picture displaying available?
	4		*	*	2: Boldface available?
	4		*	*	3: Italic available?
	4		*	*	4: Fixed-space font available?
	6		*	*	5: Sound effects available?
	4		*	*	7: Timed keyboard input available?
4	1				Base of high memory (byte address)
6	1				Initial value of program counter (byte address)
	6				Packed address of initial "main" routine
8	1				Location of dictionary (byte address)
A	1				Location of object table (byte address)

C	1				Location of global variables table (byte address)
E	1				Base of static memory (byte address)
10	1				*Flags 2:*
	1	*	*	*	Bit 0: Set when transcripting is on
	3	*		*	1: Game sets to force printing in fixed-pitch font
	6	*	*		2: Int sets to request status line redraw: game clears when it complies with this.
	5		*	*	3: If set, game wants to use pictures
	5		*	*	4: If set, game wants to use the UNDO opcodes
	5		*	*	5: If set, game wants to use a mouse
	5				6: If set, game wants to use colours
	5		*	*	7: If set, game wants to use sound effects
	6		*	*	8: If set, game wants to use menus
					(For bits 3,4,5,7 and 8, Int clears again if it cannot provide the requested effect.)
18	2				Location of abbreviations table (byte address)
1A	3+				Length of file (see note)
1C	3+				Checksum of file
1E	4		*	*	Interpreter number
1F	4		*	*	Interpreter version
Hex	**V**	**Dyn**	**Int**	**Rst**	**Contents**
20	4		*	*	Screen height (lines): 255 means "infinite"
21	4		*	*	Screen width (characters)
22	5		*	*	Screen width in units
24	5		*	*	Screen height in units
26	5		*	*	Font width in units (defined as width of a '0')
	6		*	*	Font height in units
27	5		*	*	Font height in units
	6		*	*	Font width in units (defined as width of a '0')
28	6				Routines offset (divided by 8)
2A	6				Static strings offset (divided by 8)
2C	5		*	*	Default background colour
2D	5		*	*	Default foreground colour
2E	5				Address of terminating characters table (bytes)
30	6		*		Total width in pixels of text sent to output stream 3
32	1		*	*	Standard revision number
34	5				Alphabet table address (bytes), or 0 for default
36	5				Header extension table address (bytes)

Some early Version 3 files do not contain length and checksum data, hence the notation **3+**.

11.1.1

It is illegal for a game to alter those fields not marked as "Dyn". An interpreter is therefore free to store values of such fields in its own variables.

11.1.2

The state of the transcription bit (bit 0 of Flags 2) can be changed directly by the game to turn transcribing on or off (see **S** 7.3, **S** 7.4). The interpreter must also alter it if stream 2 is turned on or off, to ensure that the bit always reflects the true state of transcribing. Note that the interpreter ensures that its value survives a restart or restore.

11.1.3

Infocom used the interpreter numbers:

1	DECSystem-20	5	Atari ST	9	Apple IIc
2	Apple IIe	6	IBM PC	10	Apple IIgs
3	Macintosh	7	Commodore 128	11	Tandy Color
4	Amiga	8	Commodore 64		

(The DECSystem-20 was Infocom's own in-house mainframe.) An interpreter should choose the interpreter number most suitable for the machine it will run on. In Versions up to 5, the main consideration is that the behaviour of 'Beyond Zork' depends on the interpreter number (in terms of its usage of the character graphics font). In Version 6, the decision is more serious, as existing Infocom story files depend on interpreter number in many ways: moreover, some story files expect to be run only on the interpreters for a particular machine. (There are, for instance, specifically Amiga versions.)

11.1.3.1

Interpreter versions are conventionally ASCII codes for upper-case letters in Versions 4 and 5 (note that Infocom's Version 6 interpreters just store numbers here).

11.1.4

******* The use of bit 7 in 'Flags 1' to signal whether timed input is available is new in this document: see the preface.

11.1.5

******* If an interpreter obeys Revision **n.m** of this document *perfectly*, as far as anyone knows, then byte **$32** should be written with **n** and byte **$33** with **m**. If it is an earlier (non-standard) interpreter, it should leave these bytes as 0.

11.1.6

The file length stored at **$1a** is actually divided by a constant, depending on the Version, to make it fit into a header word. This constant is 2 for Versions 1 to 3, 4 for Versions 4 to 5 or 8 for Versions 6 and later.

11.1.7

The header extension table provides potentially unlimited room for further header information. It is a table of word entries, in which the initial word contains the number of words of data to follow.

11.1.7.1

If the interpreter needs to read a word which is beyond the length of the extension table, or the extension table doesn't exist at all, then the result is 0.

11.1.7.2

If the interpreter needs to write a word which is beyond the length of the extension table, or the extension table doesn't exist at all, then the result is that nothing happens.

11.1.7.3

******* Words in the header extension table have been allocated as follows:

Header extension format

Word	V	Dyn	Int	Rst	Contents
0	5				Number of further words in table
1	5		*		X-coordinate of mouse after a click
2	5		*		Y-coordinate of mouse after a click
3	5				Unicode translation table address (optional)

Remarks

In the Infocom period, the larger Version 3 story files would not entirely fit on a single Atari 800 disc (though they would fit on a single Apple II, or a single PC disc). Atari versions were therefore made which were identical to the normal ones except for having Flags 1 bit 2 set, and were divided into the resident part on one disc and the rest on another. (This discovery was announced by Stefan Jokisch on 26 August 1997 and sees the end of one of the very few Z-machine mysteries left when Standard 1.0 was first published.)

See the "Infocom fact sheet" for numbers and letters of the known interpreters shipped by Infocom. Interpreter versions are conventionally the upper case letters in sequence (A, B, C, ...). At present most ports of **Zip** use interpreter number 6, and most of **ITF** use number 2.

The unusual behaviour of 'Beyond Zork' concerns its character graphics: see the remarks to **S** 16.

The Macintosh story file for 'Zork Zero' erroneously does not set the pictures bit (Flags 2, bit 3).

Contents / Preface / Overview

Section 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9 / 10 / 11 / 12 / 13 / 14 / 15 / 16

Appendix A / B / C / D / E / F

---

12. The object table

12.1

The object table is held in dynamic memory and its byte address is stored in the word at **$0a** in the header. (Recall that objects have flags attached called attributes, numbered from 0 upward, and variables attached called properties, numbered from 1 upward. An object need not provide every property.)

12.2

The table begins with a block known as the property defaults table. This contains 31 words in Versions 1 to 3 and 63 in Versions 4 and later. When the game attempts to read the value of property n for an object which does not provide property n, the n-th entry in this table is the resulting value.

12.3

Next is the object tree. Objects are numbered consecutively from 1 upward, with object number 0 being used to mean "nothing" (though there is formally no such object). The table consists of a list of entries, one for each object.

12.3.1

In Versions 1 to 3, there are at most 255 objects, each having a 9-byte entry as follows:

```
   the 32 attribute flags     parent     sibling     child   properties
   ---32 bits in 4 bytes---   ---3 bytes------------------  ---2 bytes--
```

parent, **sibling** and **child** must all hold valid object numbers. The **properties** pointer is the byte address of the list of properties attached to the object. Attributes 0 to 31 are flags (at any given time, they are either on (1) or off (0)) and are stored topmost bit first: e.g., attribute 0 is stored in bit 7 of the first byte, attribute 31 is stored in bit 0 of the fourth.

12.3.2

In Version 4 and later, there are at most 65535 objects, each having a 14-byte entry as follows:

```
   the 48 attribute flags     parent    sibling   child     properties
  ---48 bits in 6 bytes---   ---3 words, i.e. 6 bytes----  ---2 bytes--
```

12.4

Each object has its own property table. Each of these can be anywhere in dynamic memory (indeed, a game can legally change an object's properties table address in play, provided the new address points to another valid properties table). The header of a property table is as follows:

```
   text-length     text of short name of object
  -----byte----   --some even number of bytes---
```

where the **text-length** is the number of 2-byte words making up the text, which is stored in the usual format. (This means that an object's short name is limited to 765 Z-characters.) After the header, the properties are listed in descending numerical order. (This order is essential and is not a matter of convention.)

12.4.1

In Versions 1 to 3, each property is stored as a block

```
   size byte     the actual property data
                ---between 1 and 8 bytes--
```

where the **size byte** is arranged as 32 times the number of data bytes minus one, plus the property number. A property list is terminated by a size byte of 0. (It is otherwise illegal for a size byte to be a multiple of 32.)

12.4.2

In Versions 4 and later, a property block instead has the form

```
   size and number       the actual property data
  --1 or 2 bytes---     --between 1 and 64 bytes--
```

The property number occupies the bottom 6 bits of the first size byte.

12.4.2.1

If the top bit of the size byte is set, then there is a second size byte. The bottom six bits contain the property data length (counting in bytes). The seventh bit is undetermined (it is set in Infocom's datafiles, and clear in Inform's).

12.4.2.1.1

******* A value of 0 in the bottom six bits of the second byte should be interpreted as a length of 64. (Inform can compile such properties.)

12.4.2.2

Otherwise, if bit 6 of the size byte is set then the length is 2, and if it is clear then the length is 1.

12.5

It is the game's responsibility to keep the object tree well-founded: the interpreter is not required to check. "Well-founded" means the following:

(a) An object with a sibling also has a parent.

(b) An object is the parent of exactly those objects in the sibling list of its child.

(c) Each object can be given a level n, such that parentless objects have level 0 and all children of a level n object have level n+1.

Remarks

The largest valid object number is not directly stored anywhere in the Z-machine. Utility programs like **Infodump** deduce this number by assuming that, initially, the object entries end where the first property table begins.

Infocom's 'Sherlock' contains a bug making it try to set and clear attribute 48.

The reason why the second property size byte needs to have top bit set is that the size field must be parsable either forwards or backwards -- the Z-machine needs to be able to reconstruct the length of a property given only the address of the first byte of its data. (There are very many (e.g. 2000) property entries in a story file, so optimising size into one byte most of the time is worthwhile.)

Bit 6 in the second byte is presently wasted, owing to a misunderstanding by Inform (which always sets the bit: Infocom had always left it clear). This is a pity as it could be used to allow up to 128 bytes of property data. But such a change now would cause all existing Inform-compiled games to fail.

Inform can only construct well-founded object trees as the initial game state, but it is easy to compile sequences of code like "move red box to blue box" followed by "move blue box to red box" which leave the object tree in an ill-founded state. (The Inform library protects the standard object-movement verbs against this.)

Contents / Preface / Overview

Section 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9 / 10 / 11 / 12 / 13 / 14 / 15 / 16

Appendix A / B / C / D / E / F

13. The dictionary and lexical analysis

1

The dictionary table is held in static memory and its byte address is stored in the word at **$08** in the header.

2

The table begins with a short header:

```
  n     list of keyboard input codes   entry-length  number-of-entries
 byte   ------n bytes-----------------      byte         2-byte word
```

The keyboard input codes are "word-separators": typically (and under Inform mandatorily) these are the ZSCII codes for full stop, comma and double-quote. Note that a space character (32) should never be a word-separator. The "entry length" is the length of each word's entry in the dictionary table. (It must be at least 4 in Versions 1 to 3, and at least 6 in later Versions.)

2.1

Note that the word-separators table can only contain codes which are defined in ZSCII for both input and output.

3

In Versions 1 to 3, each word has an entry in the form

```
   encoded text of word        bytes of data
  ------- 4 bytes ------   (entry length-4) bytes
```

The interpreter ignores the bytes of data (presumably the game's parser will use them). The encoded text contains 6 Z-characters (it is always padded out with Z-character 5's to make up 4 bytes: see **S** 3). The text may include spaces or other word-separators (though, if so, the interpreter will never match any text to the dictionary word in question: surprisingly, this can be useful and is a trick used in the Inform library).

4

In Versions 4 and later, the encoded text has 6 bytes and always contains 9 Z-characters.

5

The word entries follow immediately after the dictionary header and must be given in numerical order of the encoded text (when the encoded text is regarded as a 32 or 48-bit binary number with most-significant byte first). It must not contain two entries with the same encoded text.

6

Lexical analysis takes place in two circumstances: on request of a **tokenise** opcode (in which case it can use any dictionary table it likes, in the format above) and during acceptance of a game command (in which case the standard dictionary is used).

6.1

First, the text is broken up into words. Spaces divide up words and are otherwise ignored. Word separators also divide words, but each one of them is considered a word in its own right. Thus, the erratically-spaced text "fred,go fishing" is divided into four words:

```
fred / , / go / fishing
```

6.2

Each word is then encoded as a Z-machine string in dictionary form, and searched for in the dictionary.

6.3

A "parse table" is then written, recording the number of words, the length and position of each word and the dictionary address of each word which is recognised. For the format, see the **read** opcode.

Remarks

Usually (under Inform, mandatorily) there are three bytes of data in the word entries, so that dictionary entry lengths are 7 and 9 in the early and late Z-machine, respectively.

It is essential that dictionary entries are in numerical order of the bytes of encrypted text so that interpreters can search the dictionary efficiently (e.g. by a binary-chop algorithm). Because the letters in A0 are in alphabetical order, because the bits are ordered in the right way and because the pad character 5 is less than the values for the letters, the numerical ordering corresponds to normal English alphabetical order for ordinary words. (For instance "an" comes before "anaconda".)

Both Infocom and Inform-compiled games contain words whose initial character is not a letter (for instance, "#record").

Linards Ticmanis reports that some of Infocom's interpreters convert question marks to spaces before lexical analysis. This is *not* Standard behaviour. (Thus, typing "What is a grue?" into 'Zork I' no longer works: the player must type "What is a grue" instead.)

Contents / Preface / Overview

Section 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9 / 10 / 11 / 12 / 13 / 14 / 15 / 16

Appendix A / B / C / D / E / F

---

14. Complete table of opcodes

Two-operand opcodes 2OP

St	Br	Opcode	Hex	V	Inform name and syntax	Link
		------	0	---	---	
	*	2OP:1	1		je a b ?(label)	**je**
	*	2OP:2	2		jl a b ?(label)	**jl**
	*	2OP:3	3		jg a b ?(label)	**jg**
	*	2OP:4	4		dec_chk (variable) value ?(label)	**dec_chk**
	*	2OP:5	5		inc_chk (variable) value ?(label)	**inc_chk**
	*	2OP:6	6		jin obj1 obj2 ?(label)	**jin**
	*	2OP:7	7		test bitmap flags ?(label)	**test**
*		2OP:8	8		or a b -> (result)	**or**
*		2OP:9	9		and a b -> (result)	**and**
	*	2OP:10	A		test_attr object attribute ?(label)	**test_attr**
		2OP:11	B		set_attr object attribute	**set_attr**
		2OP:12	C		clear_attr object attribute	**clear_attr**
		2OP:13	D		store (variable) value	**store**
		2OP:14	E		insert_obj object destination	**insert_obj**
*		2OP:15	F		loadw array word-index -> (result)	**loadw**
*		2OP:16	10		loadb array byte-index -> (result)	**loadb**
*		2OP:17	11		get_prop object property -> (result)	**get_prop**
*		2OP:18	12		get_prop_addr object property -> (result)	**get_prop_addr**
*		2OP:19	13		get_next_prop object property -> (result)	**get_next_prop**
*		2OP:20	14		add a b -> (result)	**add**

*		2OP:21	15		sub a b -> (result)	**sub**
*		2OP:22	16		mul a b -> (result)	**mul**
*		2OP:23	17		div a b -> (result)	**div**
*		2OP:24	18		mod a b -> (result)	**mod**
*		2OP:25	19	4	call_2s routine arg1 -> (result)	**call_2s**
		2OP:26	1A	5	call_2n routine arg1	**call_2n**
		2OP:27	1B	5	set_colour foreground background	**set_colour**
				6	set_colour foreground background window	**set_colour**
		2OP:28	1C	5/6	throw value stack-frame	**throw**
		------	1D	---	---	
		------	1E	---	---	
		------	1F	---	---	

Opcode numbers 32 to 127: other forms of 2OP with different types.

One-operand opcodes 1OP

St	Br	Opcode	Hex	V	Inform name and syntax	Link
	*	1OP:128	0		jz a ?(label)	**jz**
*	*	1OP:129	1		get_sibling object -> (result) ?(label)	**get_sibling**
*	*	1OP:130	2		get_child object -> (result) ?(label)	**get_child**
*		1OP:131	3		get_parent object -> (result)	**get_parent**
*		1OP:132	4		get_prop_len property-address -> (result)	**get_prop_len**
		1OP:133	5		inc (variable)	**inc**
		1OP:134	6		dec (variable)	**dec**
		1OP:135	7		print_addr byte-address-of-string	**print_addr**
*		1OP:136	8	4	call_1s routine -> (result)	**call_1s**
		1OP:137	9		remove_obj object	**remove_obj**
		1OP:138	A		print_obj object	**print_obj**
		1OP:139	B		ret value	**ret**
		1OP:140	C		jump ?(label)	**jump**
		1OP:141	D		print_paddr packed-address-of-string	**print_paddr**
*		1OP:142	E		load (variable) -> (result)	**load**
*		1OP:143	F	1/4	not value -> (result)	**not**
				5	call_1n routine	**call_1n**

Opcode numbers 144 to 175: other forms of 1OP with different types.

Zero-operand opcodes 0OP

St	Br	Opcode	Hex	V	Inform name and syntax	Link
		0OP:176	0		rtrue	**rtrue**

		0OP:177	1		rfalse	**rfalse**
		0OP:178	2		print (literal-string)	**print**
		0OP:179	3		print_ret (literal-string)	**print_ret**
		0OP:180	4	1/-	nop	**nop**
	*	0OP:181	5	1	save ?(label)	**save**
				4	save -> (result)	**save**
				5	[illegal]	
	*	0OP:182	6	1	restore ?(label)	**restore**
				4	restore -> (result)	**restore**
				5	[illegal]	
		0OP:183	7		restart	**restart**
		0OP:184	8		ret_popped	**ret_popped**
		0OP:185	9	1	pop	**pop**
*				5/6	catch -> (result)	**catch**
		0OP:186	A		quit	**quit**
		0OP:187	B		new_line	**new_line**
		0OP:188	C	3	show_status	**show_status**
				4	[illegal]	
	*	0OP:189	D	3	verify ?(label)	**verify**
		0OP:190	E	5	[first byte of extended opcode]	**extended**
	*	0OP:191	F	5/-	piracy ?(label)	**piracy**

Opcode numbers 192 to 223: VAR forms of 2OP:0 to 2OP:31.

Variable-operand opcodes VAR

St	Br	Opcode	Hex	V	Inform name and syntax	Link
*		VAR:224	0	1	call routine ...0 to 3 args... -> (result)	**call**
				4	call_vs routine ...0 to 3 args... -> (result)	**call_vs**
		VAR:225	1		storew array word-index value	**storew**
		VAR:226	2		storeb array byte-index value	**storeb**
		VAR:227	3		put_prop object property value	**put_prop**
		VAR:228	4	1	sread text parse	**sread**
				4	sread text parse time routine	**sread**
*				5	aread text parse time routine -> (result)	**aread**
		VAR:229	5		print_char output-character-code	**print_char**
		VAR:230	6		print_num value	**print_num**
*		VAR:231	7		random range -> (result)	**random**
		VAR:232	8		push value	**push**

		VAR:233	9	1	pull (variable)	pull
*				6	pull stack -> (result)	pull
		VAR:234	A	3	split_window lines	split_window
		VAR:235	B	3	set_window window	set_window
*		VAR:236	C	4	call_vs2 routine ...0 to 7 args... -> (result)	call_vs2
		VAR:237	D	4	erase_window window	erase_window
		VAR:238	E	4/-	erase_line value	erase_line
				6	erase_line pixels	erase_line
		VAR:239	F	4	set_cursor line column	set_cursor
				6	set_cursor line column window	set_cursor
		VAR:240	10	4/6	get_cursor array	get_cursor
		VAR:241	11	4	set_text_style style	set_text_style
		VAR:242	12	4	buffer_mode flag	buffer_mode
		VAR:243	13	3	output_stream number	output_stream
				5	output_stream number table	output_stream
				6	output_stream number table width	output_stream
		VAR:244	14	3	input_stream number	input_stream
		VAR:245	15	5/3	sound_effect number effect volume routine	sound_effect
*		VAR:246	16	4	read_char 1 time routine -> (result)	read_char
*	*	VAR:247	17	4	scan_table x table len form -> (result)	scan_table
*		VAR:248	18	5/6	not value -> (result)	not
		VAR:249	19	5	call_vn routine ...up to 3 args...	call_vn
		VAR:250	1A	5	call_vn2 routine ...up to 7 args...	call_vn2
		VAR:251	1B	5	tokenise text parse dictionary flag	tokenise
		VAR:252	1C	5	encode_text zscii-text length from coded-text	encode_text
		VAR:253	1D	5	copy_table first second size	copy_table
		VAR:254	1E	5	print_table zscii-text width height skip	print_table
	*	VAR:255	1F	5	check_arg_count argument-number	check_arg_count

Extended opcodes EXT

St	Br	Opcode	Hex	V	Inform name and syntax	Link
*		EXT:0	0	5	save table bytes name -> (result)	save
*		EXT:1	1	5	restore table bytes name -> (result)	restore
*		EXT:2	2	5	log_shift number places -> (result)	log_shift
*		EXT:3	3	5/-	art_shift number places -> (result)	art_shift
*		EXT:4	4	5	set_font font -> (result)	set_font
		EXT:5	5	6	draw_picture picture-number y x	draw_picture

	*	EXT:6	6	6	picture_data picture-number array ?(label)	picture_data
		EXT:7	7	6	erase_picture picture-number y x	erase_picture
		EXT:8	8	6	set_margins left right window	set_margins
*		EXT:9	9	5	save_undo -> (result)	save_undo
*		EXT:10	A	5	restore_undo -> (result)	restore_undo
		EXT:11	B	5/*	print_unicode char-number	print_unicode
		EXT:12	C	5/*	check_unicode char-number -> (result)	check_unicode
		-------	D	---	---	
		-------	E	---	---	
		-------	F	---	---	
		EXT:16	10	6	move_window window y x	move_window
		EXT:17	11	6	window_size window y x	window_size
		EXT:18	12	6	window_style window flags operation	window_style
*		EXT:19	13	6	get_wind_prop window property-number -> (result)	get_wind_prop
		EXT:20	14	6	scroll_window window pixels	scroll_window
		EXT:21	15	6	pop_stack items stack	pop_stack
		EXT:22	16	6	read_mouse array	read_mouse
		EXT:23	17	6	mouse_window window	mouse_window
	*	EXT:24	18	6	push_stack value stack ?(label)	push_stack
		EXT:25	19	6	put_wind_prop window property-number value	put_wind_prop
		EXT:26	1A	6	print_form formatted-table	print_form
	*	EXT:27	1B	6	make_menu number table ?(label)	make_menu
		EXT:28	1C	6	picture_table table	picture_table

14.1

This table contains all 119 opcodes and, taken with the dictionary in **S** 15, describes exactly what each should do. In addition, it lists which opcodes are actually used in the known Infocom story files, and documents the Inform assembly language syntax.

14.2

Formally, it is illegal for a game to contain an opcode not specified for its version. An interpreter should normally halt with a suitable message.

14.2.1

However, extended opcodes in the range EXT:29 to EXT:255 should be simply ignored (perhaps with a warning message somewhere off-screen).

14.2.2

******* EXT:11 and EXT:12 are opcodes newly added to Standard 1.0 and which can be generated in code compiled by Inform 6.12 or later. EXT:13 to EXT:15, and EXT:29 to EXT:127, are reserved for future versions of this document to specify.

14.2.3

Designers who wish to create their own "new" opcodes, for one specific game only, are asked to use opcode numbers in the range EXT:128 to EXT:255. It is easy to modify Inform to name and assemble such opcodes. (Of course the game will then have to be circulated with a suitably modified interpreter to run it.)

14.2.4

Interpreter-writers should ideally make this easy by providing a routine which is called if EXT:128 to EXT:255 are found, so that the minimum possible modification to the interpreter is needed.

Reading the opcode tables

The two columns "St" and "Br" (store and branch) mark whether an instruction stores a result in a variable, and whether it must provide a label to jump to, respectively.

The "Opcode" is written **TYPE:Decimal** where the **TYPE** is the operand count (2OP, 1OP, 0OP or VAR) or else EXT for two-byte opcodes (where the first byte is (decimal) 190). The decimal number is the lowest possible decimal opcode value. The hex number is the opcode number within each **TYPE**.

The "V" column gives the Version information. If nothing is specified, the opcode is as stated from Version 1 onwards. Otherwise, it exists only from the version quoted onwards. Before this time, its use is illegal. Some opcodes change their meanings as the Version increases, and these have more than one line of specification. Others become illegal again, and these are marked **[illegal]**.

In a few cases, the Version is given as "3/4" or some such. The first number is the Version number whose specification the opcode belongs to, and the second is the earliest Version in which the opcode is known actually to be used in an Infocom-produced story file. A dash means that it seems never to have been used (in any of Versions 1 to 6). The notation "5/*****" means that the opcode was introduced in this Standards document long after the Infocom era.

The table explicitly marks opcodes which do not exist in any version of the Z-machine as **------**: in addition, none of the extended set of codes after EXT:28 were ever used.

Inform assembly language

This section documents Inform 6 assembly language, which is richer than that of Inform 5. The Inform 6 assembler can generate every legal opcode and automatically sets any consequent header bits (for instance, a usage of **set_colour** will set the "colours needed" bit).

One way to get a picture of Inform assembly language is to compile a short program with tracing switched on (using the **-a** or **-t** switches).

1. An Inform statement beginning with an **@** is sent directly to the assembler. In the syntax below, **(variable)** and **(result)** must be variables (or **sp**, a special variable name available only in assembly language, and meaning the stack pointer); **(label)** a label (not a routine name). **(literal-string)** must be literal text in quotation marks "thus". **routine** should be the name of a routine (this assembles to its packed address). Otherwise any Inform constant term (such as **'/'** or **'beetle'**) can be given as an operand.

2. It is optional, but sensible, to place a **->** sign before a store-variable. For example, in

```
    @mul a 56 -> sp;
```

("multiply variable **a** by 56, and put the result on the stack") the **->** can be omitted, but should be included for clarity.

3. A label to branch to should be prefaced with a question mark **?**, as in

```
    @je a b ?Equal;      ! Branch to Equal if a == b
```

(If the question mark is omitted, the branch is compiled in the short form, which will only work for very nearby labels and is very seldom useful in code written by hand.) Note that the effect of any branch instruction can be negated using a tilde ~:

```
    @je a b ?~Different; ! Branch to Different if a ~= b
```

4. Labels are assembled using full stops:

```
    .MyLabel;
```

All branches must be to such a label within the same routine. (The Inform assembler imposes the same-routine restriction.)

5. Most operands are assembled in the obvious way: numbers and constant values (like characters) as numbers, variables as variables, **sp** as the value on top of the stack. There are two exceptions. "Call" opcodes expect as first operand the name of a routine to call:

```
    @call_1n MyRoutine;
```

but one can also give an indirect address, as a constant or variable, using square brackets:

```
    @call_1n [x];        ! Call routine whose address is in x
```

Secondly, seven Z-machine opcodes access variables but by their numbers: thus one should write, say, the constant 0 instead of the variable **sp**. This is inconvenient, so the Inform assembler accepts variable names instead. The operands affected are those marked as **(variable)** in the syntax chart; Inform translates the variable name as a "small constant" operand with that variable's number as value. The affected opcodes are:

```
    inc,  dec,  inc_chk,  dec_chk,  store,  pull,  load.
```

This is useful, but there is another possibility, of genuinely giving a variable operand. The Inform notation for this involves square brackets again:

```
    @inc frog;          ! Increment var "frog"
    @inc [frog];        ! Increment var whose number is in "frog"
```

Infocom story files often use such instructions.

6. The Inform assembler is also written with possible extensions to the Z-machine instruction set in mind. (Of course these can only work if a customised interpreter is used.) Simply give a specification in double-quotes where you would normally give the opcode name. For example,

```
    @"1OP:4S" 34 -> i;
    @get_prop_len 34 -> i;
```

are equivalent instructions, since **get_prop_len** is instruction 4 in the 1OP (one-operand) set, and is a Store opcode. The syntax is:

```
    "  0OP       :  decimal-number  flags  "        (range 0 to 15)
       1OP                                                 0    15
       2OP                                                 0    15
       VAR                                                 32   63
       VAR_LONG                                            32   63
       EXT                                                 0    255
       EXT_LONG                                            0    255
```

(**EXT_LONG** is a logical possibility but has not been used in the Z-machine so far: the assembler provides it in case it might be useful in future.) The possible flags are:

```
       S    Store opcode
       B    Branch opcode
       T    Text in-line instead of operands
            (as with "print" and "print_ret")
       I    "Indirect addressing": first operand is a (variable)
       Fnn  Set bit nn in Flags 2 (signalling to the interpreter that an
            unusual feature has been called for): the number is in decimal
```

For example,

```
    "EXT:128BSF14"
```

is an exotic new opcode, number 128 in the extended range, which is both Branch and Store, and the assembly of which causes bit 14 to be set in "Flags 2". See **S** 14.2 below for rules on how to number newly created opcodes.

Remarks

The opcodes EXT:5 to EXT:8 were very likely in Infocom's own Version 5 specification (documentary records of which are lost): they seem to have been partially implemented in existing Infocom interpreters, but do not occur in any existing Version 5 story file. They are here left unspecified.

The notation "5/3" for **sound_effect** is because this plainly Version 5 feature was used also in one solitary Version 3 game, 'The Lurking Horror' (the sound version of which was the last Version 3 release, in September 1987).

The 2OP opcode 0 was possibly intended for setting break-points in debugging (and may be used for this again). It was not **nop**.

read_mouse and **make_menu** are believed to have been used only in 'Journey' (based on a check of 11 Version 6 story files). **picture_table** is used once by 'Shogun' and several times by 'Zork Zero'.
