The Z-Machine Standards Document

Version 1.0

by Graham Nelson

with

the Z-Machine Common Save-File Format Standard **"Quetzal"** v1.3b by Martin Frost,
the IF Resource Collection Format Standard **"Blorb"** v1.1 by Andrew Plotkin, and
the **Proposal for a Z-Machine Standard v1.1** (draft 6) by Kevin Bracey & Jason C. Penney

(*this PDF compiled by Peer Schaefer; error-reports, suggestions etc. to:* peerschaefer@gmx.net)

Please notice that this document contains some references to the former Interactive Fiction Archive at ftp.gmd.de*, which has moved and is now (June 2002) reachable at* ftp.if-archive.org

The Z-Machine Standards Document

Version 1.0

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 They do not correspond to Infocom's unpublished opcode names.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

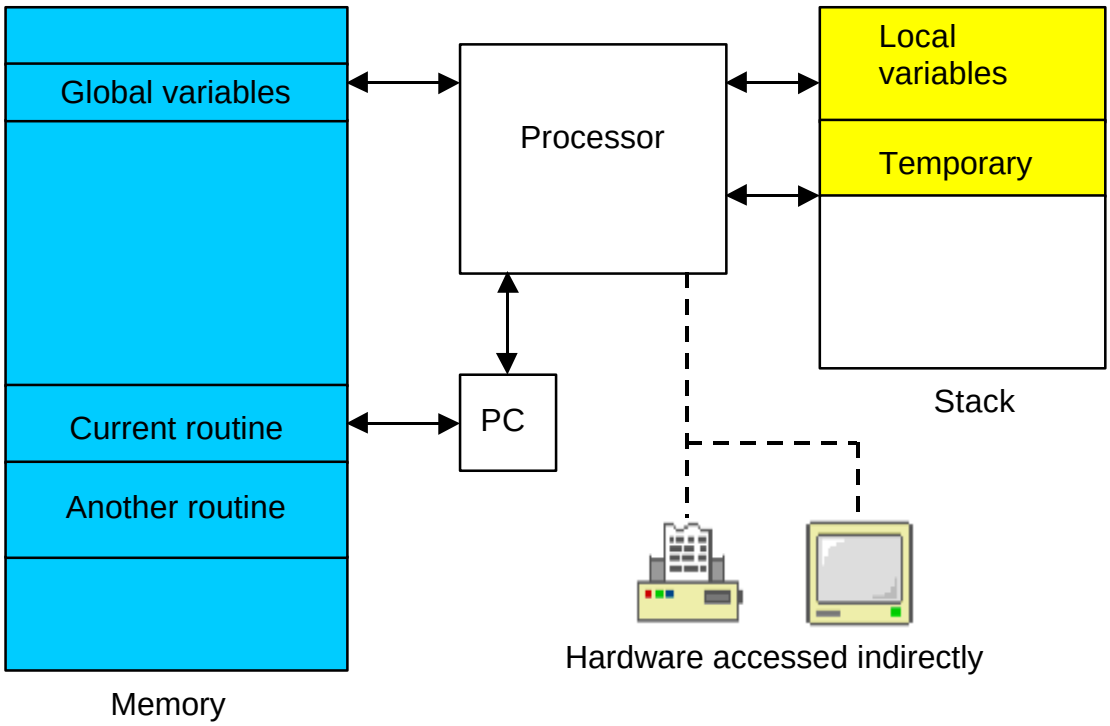
15 November 1995

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

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.

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	(kilobytes)

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.

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 Z-character**					**second Z-character**					**third Z-character**				

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 - character																								
current	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
A0	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
A1	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A2		^	0	1	2	3	4	5	6	7	8	9	.	,	!	?	_	#	‘	“	/	\	-	:	(	)

(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):

	6	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
A2		0	1	2	3	4	5	6	7	8	9	.	,	!	?	_	#	‘	“	/	\	<	-	:	(	)

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:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
$20		!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
$40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
$60	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Note that code **$23** (35 decimal) is a hash mark, not a pound sign. (Code **$7c** (124 decimal) is a vertical stroke.)

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 Con-Script 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.

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 cur-

rent 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.

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.

* The ftp.gmd.de is obsolete now. The if-archive is currently at ftp.ifarchive.org (*Peer Schaefer*, June 2002)

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.

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).

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 (bit 7) of the first size byte is set, then there are two size-and-number bytes as follows. In the first byte, bits 0 to 5 contain the property number; bit 6 is undetermined (it is clear in all Infocom or Inform story files); bit 7 is set. In the second byte, bits 0 to 5 contain the property data length, counting in bytes; bit 6 is undetermined (it is set in Infocom story files, but clear in Inform ones); bit 7 is always set.

12.4.2.1.1

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

12.4.2.2

If the top bit (bit 7) of the first size byte is clear, then there is only one size-and-number byte. Bits 0 to 5 contain the property number; bit 6 is either clear to indicate a property data length of 1, or set to indicate a length of 2; bit 7 is clear.

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, which is a pity as it could be used to allow up to 128 bytes of property data. But such a change would cause Infocom's story files to fail (since they set this bit, unlike Inform story files).

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.)

13. The dictionary and lexical analysis

13.1.

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

13.2.

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.)

13.2.1

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

13.3

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).

13.4.

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

13.5.

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.

13.6.

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).

13.6.1.

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
```

13.6.2.

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

13.6.3.

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.)

14. Complete table of opcodes

2OP / 1OP / 0OP / VAR / EXT

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 op-

code 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'.

15. Dictionary of opcodes

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

15.1

The dictionary below is alphabetical and includes entries on every opcode listed in the table above, as well as brief notes on a few opcodes once thought to exist but now disproved.

15.2

The Z-machine has the same concept of "table" (as an internal data structure) as Inform. Specifically, a table is an array of words (in dynamic or static memory) of which the initial entry is the number of subsequent words in the table. For example, a table with three entries occupies 8 bytes, arranged as the words 3, x, y, z.

15.3

In all cases below where one operand is supposed to be in a particular range, behaviour is undefined if it is not. For instance an interpreter complies with the Standard even if it crashes when an illegal object number (including 0) is given for an object operand. However, see **S** A for guidelines on detecting and dealing with errors.

---

add

2OP:20 14 add a b -> (result)

Signed 16-bit addition.

---

and

2OP:9 9 and a b -> (result)

Bitwise AND.

---

aread

This is the Inform name for the keyboard-reading opcode under Version 5 and later. (Inform calls the same opcode **sread** under Versions 3 and 4.) See **read** for the specification.

---

art_shift

EXT:3 3 5/- art_shift number places -> (result)

Does an arithmetic shift of **number** by the given number of places, shifting left (i.e. increasing)

if places is positive, right if negative. In a right shift, the sign bit is preserved as well as being shifted on down. (The alternative behaviour is **log_shift**.)

---

buffer_mode

VAR:242 12 4 buffer_mode flag

If set to 1, text output on the lower window in stream 1 is buffered up so that it can be word-wrapped properly. If set to 0, it isn't.

In Version 6, this opcode is redundant (the "buffering" window attribute can be set instead). It is used twice in each of Infocom's Version 6 story files, in the **$verify** routine. **Frotz** responds by setting the current window's "buffering" attribute, while Infocom's own interpreters respond by doing nothing. This standard leaves the result of **buffer_mode** undefined in Version 6.

---

call

VAR:224 0 1 call routine ...up to 3 args... -> (result)

The only call instruction in Version 3, Inform reads this as **call_vs** in higher versions: it calls the routine with 0, 1, 2 or 3 arguments as supplied and stores the resulting return value. (When the address 0 is called as a routine, nothing happens and the return value is false.)

---

call_1n

1OP:143 F 5 call_1n routine

Executes **routine()** and throws away result.

---

call_1s

1OP:136 8 4 call_1s routine -> (result)

Stores **routine()**.

---

call_2n

2OP:26 1A 5 call_2n routine arg1

Executes **routine(arg1)** and throws away result.

---

call_2s

2OP:25 19 4 call_2s routine arg1 -> (result)

Stores **routine(arg1)**.

---

call_vn

VAR:249 19 5 call_vn routine ...up to 3 args...

Like **call**, but throws away result.

---

call_vs

VAR:224 0 4 call_vs routine ...up to 3 args... -> (result)

See **call**.

---

call_vn2

VAR:250 1A 5 call_vn2 routine ...up to 7 args...

Call with a variable number (from 0 to 7) of arguments, then throw away the result. This (and **call_vs2**) uniquely have an extra byte of opcode types to specify the types of arguments 4 to 7. Note that it is legal to use these opcodes with fewer than 4 arguments (in which case the second byte of type information will just be **$ff**).

---

call_vs2

VAR:236 C 4 call_vs2 routine ...up to 7 args... -> (result)

See **call_vn2**.

---

catch

0OP:185 9 5/6 catch -> (result)

Opposite to **throw** (and occupying the same opcode that **pop** used in Versions 3 and 4). **catch** returns the current "stack frame".

---

check_arg_count

VAR:255 1F 5 check_arg_count argument-number

Branches if the given argument-number (counting from 1) has been provided by the routine call to the current routine. (This allows routines in Versions 5 and later to distinguish between the calls **routine(1)** and **routine(1,0)**, which would otherwise be impossible to tell apart.)

---

check_unicode

EXT:12 C 5/* check_unicode char-number -> (result)

Determines whether or not the interpreter can print, or receive from the keyboard, the given Unicode character. Bit 0 of the result should be set if and only if the interpreter can print the character; bit 1 if and only if the interpreter can receive it from the keyboard. Bits 2 to 15 are undefined.

******* This opcode will only be present in interpreters obeying Standard 1.0 or later, so story files should check the standard number of the interpreter before executing this opcode.

---

clear_attr

2OP:12 C clear_attr object attribute

Make **object** not have the attribute numbered **attribute**.

"clear_flag"

A name once used for one of the not-really-present extended Version 5 opcodes (now removed from the specification).

---

copy_table

VAR:253 1D 5 copy_table first second size

If **second** is zero, then **size** bytes of **first** are zeroed.

Otherwise **first** is copied into **second**, its length in bytes being the absolute value of **size** (i.e., **size** if **size** is positive, **-size** if **size** is negative).

The tables are allowed to overlap. If **size** is positive, the interpreter must copy either forwards or backwards so as to avoid corrupting **first** in the copying process. If **size** is negative, the interpreter must copy forwards even if this corrupts **first**. ('Beyond Zork' uses this to fill an array with spaces.)

(Version 0.2 of this document wrongly specified that if **size** is positive then copying should always run backward. This results in the player being unable to cross the river near the start of 'Journey', as the game uses **copy_table** to shuffle menu options, and the menu "Downstream, Upstream, Cross, Return" is changed to "Return, Return, Return".)

---

dec

1OP:134 6 dec (variable)

Decrement variable by 1. This is signed, so 0 decrements to -1.

---

dec_chk

2OP:4 4 dec_chk (variable) value ?(label)

Decrement variable, and branch if it is now less than the given value.

---

div

2OP:23 17 div a b -> (result)

Signed 16-bit division. Division by zero should halt the interpreter with a suitable error message.

---

draw_picture

EXT:5 5 6 draw_picture picture-number y x

Displays the picture with the given number. (y,x) coordinates (of the top left of the picture) are each optional, in that a value of zero for y or x means the cursor y or x coordinate in the current window. It is illegal to call this with an invalid picture number.

---

encode_text

VAR:252 1C 5 encode_text zscii-text length from coded-text

Translates a ZSCII word to Z-encoded text format (stored at **coded-text**), as if it were an entry in the dictionary. The text begins at **from** in the **zscii-text** buffer and is **length** characters long. (Some interpreters ignore this and keep translating until they hit a 0 character anyway, or have already filled up the 6-byte Z-encoded string.)

---

erase_line

VAR:238 E 4/6 erase_line value

Versions 4 and 5: if the value is 1, erase from the current cursor position to the end of its line in the current window. If the value is anything other than 1, do nothing.

Version 6: if the value is 1, erase from the current cursor position to the end of the its line in the current window. If not, erase the given number of pixels minus one across from the cursor (clipped to stay inside the right margin). The cursor does not move.

---

erase_picture

EXT:7 7 6 erase_picture picture-number y x

Like **draw_picture**, but paints the appropriate region to the background colour for the given window. It is illegal to call this with an invalid picture number.

---

erase_window

VAR:237 D 4 erase_window window

Erases window with given number (to background colour); or if -1 it unsplits the screen and clears the lot; or if -2 it clears the screen without unsplitting it. In cases -1 and -2, the cursor may move (see **S** 8 for precise details).

"extended"

This byte (decimal 190) is not an instruction, but indicates that the opcode is "extended": the next byte contains the number in the extended set.

---

get_child

1OP:130 2 get_child object -> (result) ?(label)

Get first object contained in given object, branching if this exists, i.e. is not **nothing** (i.e., is not 0).

---

get_cursor

VAR:240 10 4/6 get_cursor array

Puts the current cursor row into the word 0 of the given array, and the current cursor column into word 1. (The array is not a table and has no size information in its initial entry.)

---

get_next_prop

2OP:19 13 get_next_prop object property -> (result)

Gives the number of the next property provided by the quoted object. This may be zero, indicating the end of the property list; if called with zero, it gives the first property number present. It is illegal to try to find the next property of a property which does not exist, and an interpreter should halt with an error message (if it can efficiently check this condition).

---

get_parent

1OP:131 3 get_parent object -> (result)

Get parent object (note that this has no "branch if exists" clause).

---

get_prop

2OP:17 11 get_prop object property -> (result)

Read property from object (resulting in the default value if it had no such declared property). If the property has length 1, the value is only that byte. If it has length 2, the first two bytes of the property are taken as a word value. It is illegal for the opcode to be used if the property has length greater than 2, and the result is unspecified.

---

get_prop_addr

2OP:18 12 get_prop_addr object property -> (result)

Get the byte address (in dynamic memory) of the property data for the given object's property. This must return 0 if the object hasn't got the property.

---

get_prop_len

1OP:132 4 get_prop_len property-address -> (result)

Get length of property data (in bytes) for the given object's property. It is illegal to try to find the property length of a property which does not exist for the given object, and an interpreter should halt with an error message (if it can efficiently check this condition).

---

get_sibling

1OP:129 1 get_sibling object -> (result) ?(label)

Get next object in tree, branching if this exists, i.e. is not 0.

---

get_wind_prop

EXT:19 13 6 get_wind_prop window property-number -> (result)

Reads the given property of the given window (see **S** 8).

---

inc

1OP:133 5 inc (variable)

Increment variable by 1. (This is signed, so -1 increments to 0.)

---

inc_chk

2OP:5 5 inc_chk (variable) value ?(label)

Increment variable, and branch if now greater than value.

---

input_stream

VAR:244 14 3 input_stream number

Selects the current input stream.

---

insert_obj

2OP:14 E insert_obj object destination

Moves object O to become the first child of the destination object D. (Thus, after the operation the **child** of D is O, and the **sibling** of O is whatever was previously the **child** of D.) All children of O move with it. (Initially O can be at any point in the object tree; it may legally have **parent** zero.)

---

je

2OP:1 1 je a b ?(label)

Jump if **a** is equal to any of the subsequent operands. (Thus **@je a** never jumps and **@je a b** jumps if **a = b**.)

---

jg

2OP:3 3 jg a b ?(label)

Jump if **a > b** (using a signed 16-bit comparison).

---

jin

2OP:6 6 jin obj1 obj2 ?(label)

Jump if object **a** is a direct child of **b**, i.e., if **parent** of **a** is **b**.

---

jl

2OP:2 2 jl a b ?(label)

Jump if **a < b** (using a signed 16-bit comparison).

---

jump

1OP:140 C jump ?(label)

Jump (unconditionally) to the given label. (This is not a branch instruction and the operand is a 2-byte signed offset to apply to the program counter.) It is legal for this to jump into a different routine (which should not change the routine call state), although it is considered bad practice to do so and the **Txd** disassembler is confused by it.

---

jz

1OP:128 0 jz a ?(label)

Jump if **a = 0**.

---

load

1OP:142 E load (variable) -> (result)

The value of the variable referred to by the operand is stored in the result. (Inform doesn't use this; see the notes to **S** 14.)

---

loadb

2OP:16 10 loadb array byte-index -> (result)

Stores **array->byte-index** (i.e., the byte at address **array+byte-index**, which must lie in static or dynamic memory).

---

loadw

2OP:15 F loadw array word-index -> (result)

Stores **array-->word-index** (i.e., the word at address **array+2*word-index**, which must lie in static or dynamic memory).

---

log_shift

EXT:2 2 5 log_shift number places -> (result)

Does a logical shift of **number** by the given number of **places**, shifting left (i.e. increasing) if **places** is positive, right if **negative**. In a right shift, the sign is zeroed instead of being shifted on. (See also **art_shift**.)

---

make_menu

EXT:27 1B 6 make_menu number table ?(label)

Controls menus with numbers greater than 2 (i.e., it doesn't control the three system menus). If the table supplied is 0, the menu is removed. Otherwise it is a table of tables. Each table is a ZSCII string: the first item being a menu name, subsequent ones the entries.

---

mod

2OP:24 18 mod a b -> (result)

Remainder after signed 16-bit division. Division by zero should halt the interpreter with a suitable error message.

---

mouse_window

EXT:23 17 6 mouse_window window

Constrain the mouse arrow to sit inside the given window. By default it sits in window 1. Setting to -1 takes all restriction away. (The mouse clicks are not reported if the arrow is outside the window and interpreters are presumably supposed to hold the arrow there by hardware means if possible.)

---

move_window

EXT:16 10 6 move_window window y x

Moves the given window to pixels (y,x): (1,1) being the top left. Nothing actually happens (since windows are entirely notional transparencies): but any future plotting happens in the new place.

---

mul

2OP:22 16 mul a b -> (result)

Signed 16-bit multiplication.

---

new_line

0OP:187 B new_line

Print carriage return.

---

nop

0OP:180 4 1/- nop

Probably the official "no operation" instruction, which, appropriately, was never operated (in any of the Infocom datafiles): it may once have been a breakpoint.

---

not

1OP:143 F 1/4 not value -> (result)

VAR:248 18 5/6 not value -> (result)

Bitwise NOT (i.e., all 16 bits reversed). Note that in Versions 3 and 4 this is a 1OP instruction, reasonably since it has 1 operand, but in later Versions it was moved into the extended set to make room for **call_1n**.

---

or

2OP:8 8 or a b -> (result)

Bitwise OR.

---

output_stream

VAR:243 13 3 output_stream number

5 output_stream number table

6 output_stream number table width

If **stream** is 0, nothing happens. If it is positive, then that stream is selected; if negative, deselected. (Recall that several different streams can be selected at once.)

When stream 3 is selected, a **table** must be given into which text can be printed. The first word always holds the number of characters printed, the actual text being stored at bytes **table+2** onward. It is not the interpreter's responsibility to worry about the length of this table being overrun.

In Version 6, a **width** field may optionally be given: if this is non-zero, text will then be justified as if it were in the window with that number (if width is positive) or a box **-width** pixels wide (if negative). Then the table will contain not ordinary text but formatted text: see **print_form**.

---

picture_data

EXT:6 6 6 picture_data picture-number array ?(label)

Asks the interpreter for data on the picture with the given number. If the picture number is valid, a branch occurs and information is written to the array: the height in word 0, the width in word 1, in pixels. (This is an array, not a "table" with initial size information.)

Otherwise, if the picture number is zero, the interpreter writes the number of available pictures into word 0 of the array and the release number of the picture file into word 1. (Infocom's first Version 6 Amiga interpreter did not handle this case properly, and early releases of 'Zork Zero' did not use it. The feature may have been added on the MSDOS release of 'Zork Zero'.)

Otherwise, nothing happens.

---

picture_table

EXT:28 1C 6 picture_table table

Given a table of picture numbers, the interpreter may if it wishes load or unpack these pictures from disc into a cache for convenient rapid plotting later. 'Zork Zero' makes frequent use of this, for instance for its peggleboard display. Moreover, it expects rapid plotting only for those images listed in the last call to **picture_table**. In other words, any images still in the cache when **picture_table** is called can safely be thrown away. (The Amiga interpreter 6.14 uses a cache of size 5K and never caches any individual image larger than 1K.)

---

piracy

0OP:191 F 5/- piracy ?(label)

Branches if the game disc is believed to be genuine by the interpreter (which is assumed to have

some arcane way of finding out). Interpreters are asked to be gullible and to unconditionally branch.

---

pop

0OP:185 9 1 pop

Throws away the top item on the stack. (This was useful to lose unwanted routine call results in early Versions.)

---

pop_stack

EXT:21 15 6 pop_stack items stack

The given number of items are thrown away from the top of a stack: by default the system stack, otherwise the one given as a second operand.

---

print

0OP:178 2 print

Print the quoted (literal) Z-encoded string.

---

print_addr

1OP:135 7 print_addr byte-address-of-string

Print (Z-encoded) string at given byte address, in dynamic or static memory.

---

print_char

VAR:229 5 print_char output-character-code

Print a ZSCII character. The operand must be a character code defined in ZSCII for output (see **S** 3). In particular, it must certainly not be negative or larger than 1023.

---

print_form

EXT:26 1A 6 print_form formatted-table

Prints a formatted table of the kind written to output stream 3 when formatting is on. This is an elaborated version of **print_table** to cope with fonts, pixels and other impedimenta. It is a sequence of lines, terminated with a zero word. Each line is a word containing the number of characters, followed by that many bytes which hold the characters concerned.

---

print_num

VAR:230 6 print_num value

Print (signed) number in decimal.

---

print_obj

1OP:138 A print_obj object

Print short name of object (the Z-encoded string in the object header, not a property). If the object number is invalid, the interpreter should halt with a suitable error message.

---

print_paddr

1OP:141 D print_paddr packed-address-of-string

Print the (Z-encoded) string at the given packed address in high memory.

---

print_ret

0OP:179 3 print_ret

Print the quoted (literal) Z-encoded string, then print a new-line and then return true (i.e., 1).

---

print_table

VAR:254 1E 5 print_table zscii-text width height skip

Print a rectangle of text on screen spreading right and down from the current cursor position, of given **width** and **height**, from the table of ZSCII text given. (Height is optional and defaults to 1.) If a **skip** value is given, then that many characters of text are skipped over in between each line and the next. (So one could make this display, for instance, a 2 by 3 window onto a giant 40 by 40 character graphics map.)

---

print_unicode

EXT:11 B 5/* print_unicode char-number

Print a Unicode character. See **S** 3.8.5.4 and **S** 7.5 for details. The given character code must be defined in Unicode.

******* This opcode will only be present in interpreters obeying Standard 1.0 or later, so story files should check the standard number of the interpreter before executing this opcode.

---

pull

VAR:233 9 1 pull (variable)

6 pull stack -> (result)

Pulls value off a stack. (If the stack underflows, the interpreter should halt with a suitable error message.) In Version 6, the stack in question may be specified as a user one: otherwise it is the game stack.

---

push

VAR:232 8 push value

Pushes value onto the game stack.

---

push_stack

EXT:24 18 6 push_stack value stack ?(label)

Pushes the value onto the specified user stack, and branching if this was successful. If the stack overflows, nothing happens (this is not an error condition).

---

put_prop

VAR:227 3 put_prop object property value

Writes the given value to the given property of the given object. If the property does not exist for that object, the interpreter should halt with a suitable error message. If the property length is 1, then the interpreter should store only the least significant byte of the value. (For instance, storing -1 into a 1-byte property results in the property value 255.) As with **get_prop** the property length must not be more than 2: if it is, the behaviour of the opcode is undefined.

---

put_wind_prop

EXT:25 19 6 put_wind_prop window property-number value

Writes a window property (see **get_wind_prop**). This should only be used when there is no direct command (such as **move_window**) to use instead, as some such operations may have side-effects.

---

quit

0OP:186 A quit

Exit the game immediately. (Any "Are you sure?" question must be asked by the game, not the interpreter.) It is not legal to return from the main routine (that is, from where execution first begins) and this must be used instead.

---

random

VAR:231 7 random range -> (result)

If **range** is positive, returns a uniformly random number between 1 and **range**. If **range** is negative, the random number generator is seeded to that value and the return value is 0. Most interpreters consider giving 0 as **range** illegal (because they attempt a division with remainder by the **range**), but correct behaviour is to reseed the generator in as random a way as the interpreter can (e.g. by using the time in milliseconds).

(Some version 3 games, such as 'Enchanter' release 29, had a debugging verb **#random** such that typing, say, **#random 14** caused a call of **random** with -14.)

---

read

VAR:228 4 1 sread text parse

4 sread text parse time routine

5 aread text parse time routine -> (result)

(Note that Inform internally names the **read** opcode as **aread** in Versions 5 and later and **sread**

in Versions 3 and 4.)

This opcode reads a whole command from the keyboard (no prompt is automatically displayed). It is legal for this to be called with the cursor at any position on any window.

In Versions 1 to 3, the status line is automatically redisplayed first.

A sequence of characters is read in from the current input stream until a carriage return (or, in Versions 5 and later, any terminating character) is found.

In Versions 1 to 4, byte 0 of the **text-buffer** should initially contain the maximum number of letters which can be typed, minus 1 (the interpreter should not accept more than this). The text typed is reduced to lower case (so that it can tidily be printed back by the program if need be) and stored in bytes 1 onward, with a zero terminator (but without any other terminator, such as a carriage return code). (This means that if byte 0 contains n then the buffer must contain n+1 bytes, which makes it a **string** array of length n in Inform terminology.)

In Versions 5 and later, byte 0 of the **text-buffer** should initially contain the maximum number of letters which can be typed (the interpreter should not accept more than this). The interpreter stores the number of characters actually typed in byte 1 (not counting the terminating character), and the characters themselves in bytes 2 onward (not storing the terminating character). (Some interpreters wrongly add a zero byte after the text anyway, so it is wise for the buffer to contain at least n+3 bytes.)

Moreover, if byte 1 contains a positive value at the start of the input, then **read** assumes that number of characters are left over from an interrupted previous input, and writes the new characters after those already there. Note that the interpreter does not redisplay the characters left over: the game does this, if it wants to. This is unfortunate for any interpreter wanting to give input text a distinctive appearance on-screen, but 'Beyond Zork', 'Zork Zero' and 'Shogun' clearly require it. ("Just a tremendous pain in my butt" -- Andrew Plotkin; "the most unfortunate feature of the Z-machine design" -- Stefan Jokisch.)

In Version 4 and later, if the operands **time** and **routine** are supplied (and non-zero) then the routine call **routine()** is made every **time/10** seconds during the keyboard-reading process. If this routine returns true, all input is erased (to zero) and the reading process is terminated at once. (The terminating character code is 0.) The **routine** is permitted to print to the screen even if it returns false to signal "carry on": the interpreter should notice and redraw the input line so far, before input continues. (**Frotz** notices by looking to see if the cursor position is at the left-hand margin after the interrupt routine has returned.)

If input was terminated in the usual way, by the player typing a carriage return, then a carriage return is printed (so the cursor moves to the next line). If it was interrupted, the cursor is left at the rightmost end of the text typed in so far.

Next, lexical analysis is performed on the text (except that in Versions 5 and later, if **parse-buffer** is zero then this is omitted). Initially, byte 0 of the **parse-buffer** should hold the maximum number of textual words which can be parsed. (If this is n, the buffer must be at least 2 + 4*n bytes long to hold the results of the analysis.)

The interpreter divides the text into words and looks them up in the dictionary, as described in **S** 13. The number of words is written in byte 1 and one 4-byte block is written for each word, from byte 2 onwards (except that it should stop before going beyond the maximum number of words specified). Each block consists of the byte address of the word in the dictionary, if it is in the dictionary, or 0 if it isn't; followed by a byte giving the number of letters in the word; and finally a byte giving the position in the **text-buffer** of the first letter of the word.

In Version 5 and later, this is a store instruction: the return value is the terminating character

(note that the user pressing his "enter" key may cause either 10 or 13 to be returned; the author recommends that interpreters return 10). A timed-out input returns 0.

(Versions 1 and 2 and early Version 3 games mistakenly write the parse buffer length 240 into byte 0 of the parse buffer: later games fix this bug and write 59, because 2+4*59 = 238 so that 59 is the maximum number of textual words which can be parsed into a buffer of length 240 bytes. Old versions of the Inform 5 library commit the same error. Neither mistake has very serious consequences.)

(Interpreters are asked to halt with a suitable error message if the text or parse buffers have length of less than 3 or 6 bytes, respectively: this sometimes occurs due to a previous array being overrun, causing bugs which are very difficult to find.)

---

read_char

VAR:246 16 4 read_char 1 time routine -> (result)

Reads a single character from input stream 0 (the keyboard). The first operand must be 1 (presumably it was provided to support multiple input devices, but only the keyboard was ever used). **time** and **routine** are optional (in Versions 4 and later only) and dealt with as in **read** above.

---

read_mouse

EXT:22 16 6 read_mouse array

The four words in the **array** are written with the mouse y coordinate, x coordinate, button bits (low bits on the right of the mouse, rising as one looks left), and a menu word. In the menu word, the upper byte is the menu number and the lower byte is the item number (from 0). (Note that the array isn't a table and has no initial size information. The data is written to words 0 to 3 in the array.)

---

remove_obj

1OP:137 9 remove_obj object

Detach the object from its parent, so that it no longer has any parent. (Its children remain in its possession.)

---

restart

0OP:183 7 1 restart

Restart the game. (Any "Are you sure?" question must be asked by the game, not the interpreter.) The only pieces of information surviving from the previous state are the "transcribing to printer" bit (bit 0 of 'Flags 2' in the header, at address **$10**) and the "use fixed pitch font" bit (bit 1 of 'Flags 2').

In particular, changing the program start address before a restart will not have the effect of restarting from this new address.

---

restore

0OP:182 6 1 restore ?(label)

0OP:182 5 4 restore -> (result)

EXT:1 1 5 restore table bytes name -> (result)

See **save**. In Version 3, the branch is never actually made, since either the game has successfully picked up again from where it was saved, or it failed to load the save game file.

As with **restart**, the transcription and fixed font bits survive. The interpreter gives the game a way of knowing that a restore has just happened (see **save**).

******* From Version 5 it can have optional parameters as **save** does, and returns the number of bytes loaded if so. (Whether Infocom intended these options as part of Version 5 is doubtful, but it's too useful a feature to exclude from this Standard.)

If the restore fails, 0 is returned, but once again this necessarily happens since otherwise control is already elsewhere.

---

restore_undo

EXT:10 A 5 restore_undo -> (result)

Like **restore**, but restores the state saved to memory by **save_undo**. (The optional parameters of **restore** may not be supplied.) The behaviour of **restore_undo** is unspecified if no **save_undo** has previously occurred (and a game may not legally use it): an interpreter might simply ignore this.

---

ret

1OP:139 B ret value

Returns from the current routine with the value given.

---

ret_popped

0OP:184 8 ret_popped

Pops top of stack and returns that. (This is equivalent to **ret sp**, but is one byte cheaper.)

---

rfalse

0OP:177 1 rfalse

Return false (i.e., 0) from the current routine.

---

rtrue

0OP:176 0 rtrue

Return true (i.e., 1) from the current routine.

---

save

0OP:181 5 1 save ?(label)

0OP:181 5 4 save -> (result)

EXT:0 0 5 save table bytes name -> (result)

On Versions 3 and 4, attempts to save the game (all questions about filenames are asked by interpreters) and branches if successful. From Version 5 it is a store rather than a branch instruction; the store value is 0 for failure, 1 for "save succeeded" and 2 for "the game is being restored and is resuming execution again from here, the point where it was saved".

It is illegal to use this opcode within an interrupt routine (one called asynchronously by a sound effect, or keyboard timing, or newline counting).

******* The extension also has (optional) parameters, which save a region of the save area, whose address and length are in bytes, and provides a suggested filename: name is a pointer to an array of ASCII characters giving this name (as usual preceded by a byte giving the number of characters). See **S** 7.6. (Whether Infocom intended these options as part of Version 5 is doubtful, but it's too useful a feature to exclude from this Standard.)

---

save_undo

EXT:9 9 5 save_undo -> (result)

Like **save**, except that the optional parameters may not be specified: it saves the game into a cache of memory held by the interpreter. If the interpreter is unable to provide this feature, it must return -1: otherwise it returns the **save** return value.

It is illegal to use this opcode within an interrupt routine (one called asynchronously by a sound effect, or keyboard timing, or newline counting).

(This call is typically needed once per turn, in order to implement "UNDO", so it needs to be quick.)

---

scan_table

VAR:247 17 4 scan_table x table len form -> (result)

Is **x** one of the words in **table**, which is **len** words long? If so, return the address where it first occurs and branch. If not, return 0 and don't.

The **form** is optional (and only used in Version 5?): bit 8 is set for words, clear for bytes: the rest contains the length of each field in the table. (The first word or byte in each field being the one looked at.) Thus **$82** is the default.

---

scroll_window

EXT:20 14 6 scroll_window window pixels

Scrolls the given window by the given number of pixels (a negative value scrolls backwards, i.e., down) writing in blank (background colour) pixels in the new lines. This can be done to any window and is not related to the "scrolling" attribute of a window.

---

set_attr

2OP:11 B set_attr object attribute

Make **object** have the attribute numbered **attribute**.

---

set_colour

2OP:27 1B 5 set_colour foreground background

6 set_colour foreground background window

If coloured text is available, set text to be foreground-against-background. (Flush any buffered text to screen, in the old colours, first.) In version 6, the **window** argument is optional and is by default the current window. (This option is supported in Infocom's Amiga and DOS interpreters.)

(One Version 5 game uses this: 'Beyond Zork' (Paul David Doherty reports it as used "76 times in 870915 and 870917, 58 times in 871221") and from the structure of the table it clearly logically belongs in version 5.)

---

set_cursor

VAR:239 F 4 set_cursor line column

6 set_cursor line column window

Move cursor in the current window to the position (x,y) (in units) relative to (1,1) in the top left. (In Version 6 the window is supplied and need not be the current one. Also, if the cursor would lie outside the current margin settings, it is moved to the left margin of the current line.)

In Version 6, **set_cursor -1** turns the cursor off, and either **set_cursor -2** or **set_cursor -2 0** turn it back on. It is not known what, if anything, this second argument means: in all known cases it is 0.

"set_flag"

See **clear_flag**.

---

set_font

EXT:4 4 5 set_font font -> (result)

If the requested font is available, then it is chosen for the current window, and the store value is the font ID of the previous font (which is always positive). If the font is unavailable, nothing will happen and the store value is 0.

(Infocom's old interpreters did not store 0 for an unavailable font, but the feature is clearly useful and so was introduced in release 0.2 of this Standard. The opcode had an optional extra **window** operand in Version 6, but this has never been used and is now withdrawn from the Standard.)

---

set_margins

EXT:8 8 6 set_margins left right window

Sets the margin widths (in pixels) on the left and right for the given window (which are by default 0). If the cursor is overtaken and now lies outside the margins altogether, move it back to

the left margin of the current line (see **S** 8.8.3.2.2.1).

---

set_text_style

VAR:241 11 4 set_text_style style

Sets the text style to: Roman (if 0), Reverse Video (if 1), Bold (if 2), Italic (4), Fixed Pitch (8). In some interpreters (though this is not required) a combination of styles is possible (such as reverse video and bold). In these, changing to Roman should turn off all the other styles currently set.

---

set_window

VAR:235 B 3 set_window window

Selects the given window for text output.

---

show_status

0OP:188 C 3 show_status

(In Version 3 only.) Display and update the status line now (don't wait until the next keyboard input). (In theory this opcode is illegal in later Versions but an interpreter should treat it as **nop**, because Version 5 Release 23 of 'Wishbringer' contains this opcode by accident.)

---

sound_effect

VAR:245 15 5/3 sound_effect number effect volume routine

The given effect happens to the given sound number. The low byte of **volume** holds the volume level, the high byte the number of repeats. (The value 255 means "loudest possible" and "forever" respectively.) (In Version 3, repeats are unsupported and the high byte must be 0.)

Note that sound effect numbers 1 and 2 are bleeps (see **S** 9) and in these cases the other operands must be omitted. Conversely, if any of the other operands are present, the sound effect number must be 3 or higher.

The **effect** can be: 1 (prepare), 2 (start), 3 (stop), 4 (finish with).

In Versions 5 and later, the **routine** is called (with no parameters) after the sound has been finished (it has been playing in the background while the Z-machine has been working on other things). (This is used by 'Sherlock' to implement fading in and out, which explains why mysterious numbers like **$34FB** were previously thought to be to do with fading.) The routine is not called if the sound is stopped by another sound or by an effect 3 call.

See the remarks to **S** 9 for which forms of this opcode were actually used by Infocom.

In theory, **@sound_effect;** (with no operands at all) is illegal. However interpreters are asked to beep (as if the operand were 1) if possible, and in any case not to halt.

---

split_window

VAR:234 A 3 split_window lines

Splits the screen so that the upper window has the given number of lines: or, if this is zero,

unsplits the screen again. In Version 3 (only) the upper window should be cleared after the split.

In Version 6, this is supposed to roughly emulate the earlier Version 5 behaviour (see **S** 8), though the line count is in units rather than lines. (Existing Version 6 games seem to use this opcode only for bounding cursor movement. 'Journey' creates a status region which is the whole screen and then overlays it with two other windows.) The width and x-coordinates of windows 0 and 1 are not altered. A cursor remains in the same absolute screen position (which means that its y-coordinate will be different relative to the window origin, since this origin will have moved) unless this position is no longer in the window at all, in which case it is moved to the window origin (at the top left of the window).

---

sread

This is the Inform name for the keyboard-reading opcode under Versions 3 and 4. (Inform calls the same opcode **aread** in later Versions.) See **read** for the specification.

---

store

2OP:13 D store (variable) value

Set the **VARiable** referenced by the operand to **value**.

---

storeb

VAR:226 2 storeb array byte-index value

array->byte-index = value, i.e. stores the given value in the byte at address **array+byte-index** (which must lie in dynamic memory). (See **loadb**.)

---

storew

VAR:225 1 storew array word-index value

array-->word-index = value, i.e. stores the given value in the word at address **array+2*word-index** (which must lie in dynamic memory). (See **loadw**.)

---

sub

2OP:21 15 sub a b -> (result)

Signed 16-bit subtraction.

---

test

2OP:7 7 test bitmap flags ?(label)

Jump if all of the flags in bitmap are set (i.e. if **bitmap & flags == flags**).

"test_array"

See **clear_flag**. (**ITF** implements this as unconditionally false.)

---

test_attr

2OP:10 A test_attr object attribute ?(label)

Jump if **object** has **attribute**.

---

throw

2OP:28 1C 5/6 throw value stack-frame

Opposite of **catch**: resets the routine call state to the state it had when the given stack frame value was 'caught', and then returns. In other words, it returns as if from the routine which executed the **catch** which found this stack frame value.

---

tokenise

VAR:251 1B 5 tokenise text parse dictionary flag

This performs lexical analysis (see **read** above).

If a non-zero **dictionary** is supplied, it is used (if not, the ordinary game dictionary is). If the **flag** is set, unrecognised words are not written into the parse buffer and their slots are left unchanged: this is presumably so that if several **tokenise** instructions are performed in a row, each fills in more slots without wiping those filled by the others.

Parsing a user dictionary is slightly different. A user dictionary should look just like the main one but need not be alphabetically sorted. If the number of entries is given as -n, then the interpreter reads this as "n entries unsorted". This is very convenient if the table is being altered in play: if, for instance, the player is naming things.

---

verify

0OP:189 D 3 verify ?(label)

Verification counts a (two byte, unsigned) checksum of the file from **$0040** onwards (by taking the sum of the values of each byte in the file, modulo **$10000**) and compares this against the value in the game header, branching if the two values agree. (Early Version 3 games do not have the necessary checksums to make this possible.)

The interpreter may stop calculating when the file length (as given in the header) is reached. It is legal for the file to contain more bytes than this, but if so the extra bytes must all be 0, which would contribute nothing the checksum anyway. (Some story files are padded out to an exact number of virtual-memory pages using 0s.)

---

window_size

EXT:17 11 6 window_size window y x

Change size of window in pixels. (Does not change the current display.)

---

window_style

EXT:18 12 6 window_style window flags operation

Changes attributes for a given window. A bitmap of attributes is given, in which the bits are: 0 -- keep text within margins, 1 -- scroll when at bottom, 2 -- copy text to output stream 2 (the printer), 3 -- buffer text to word-wrap it between the margins of the window.

The operation, by default, is 0, meaning "set to these settings". 1 means "set the bits supplied". 2 means "clear the ones supplied", and 3 means "reverse the bits supplied" (i.e. eXclusive OR).

16. Font 3 and character graphics

16.1

The following table of 8x8 bitmaps gives a suitable appearance for font 3. The font must have a fixed pitch and characters must be printed immediately next to each other in all four directions.

```
32( ):  76543210   33(!):  76543210   34("):  76543210   35(#):  76543210
       0                  0                  0                  0       #
       1                  1                  1                  1      #
       2                  2  #               2    #             2     #
       3                  3 ##               3    ##            3    #
       4                  4#######           4#######           4   #
       5                  5 ##               5    ##            5  #
       6                  6  #               6    #             6 #
       7                  7                  7                  7#

36($):  76543210   37(%):  76543210   38(&):  76543210   39('):  76543210
       0#                 0                  0                  0
       1 #                1                  1                  1
       2  #               2                  2                  2
       3   #              3                  3                  3########
       4    #             4                  4########          4
       5     #            5                  5                  5
       6      #           6                  6                  6
       7       #          7                  7                  7

40(():  76543210   41()):  76543210   42(*):  76543210   43(+):  76543210
       0    #             0   #              0    #             0
       1    #             1   #              1    #             1
       2    #             2   #              2    #             2
       3    #             3   #              3########          3
       4    #             4   #              4                  4########
       5    #             5   #              5                  5    #
       6    #             6   #              6                  6    #
       7    #             7   #              7                  7    #

44(,):  76543210   45(-):  76543210   46(.):  76543210   47(/):  76543210
       0    #             0   #              0   #              0
       1    #             1   #              1   #              1
       2    #             2   #              2   #              2
       3    #             3   #              3   #              3   #####
       4    ####          4####              4   #####          4   #
       5    #             5   #              5                  5   #
       6    #             6   #              6                  6   #
       7    #             7   #              7                  7   #

48(0):  76543210   49(1):  76543210   50(2):  76543210   51(3):  76543210
       0                  0    #             0   #              0#
       1                  1    #             1   #              1 #
       2                  2    #             2   #              2  #
       3#####             3    #             3   #              3   #####
       4    #             4#####             4   #####          4   #
       5    #             5                  5  #               5   #
       6    #             6                  6 #                6   #
       7    #             7                  7#                 7   #
```

```
52(4):  76543210
       0       #
       1      #
       2     #
       3#####
       4    #
       5    #
       6    #
       7    #

53(5):  76543210
       0    #
       1    #
       2    #
       3    #
       4#####
       5     #
       6      #
       7       #

54(6):  76543210
       0########
       1########
       2########
       3########
       4########
       5########
       6########
       7########

55(7):  76543210
       0########
       1########
       2########
       3########
       4########
       5
       6
       7

56(8):  76543210
       0
       1
       2
       3########
       4########
       5########
       6########
       7########

57(9):  76543210
       0#####
       1#####
       2#####
       3#####
       4#####
       5#####
       6#####
       7#####

58(:):  76543210
       0   #####
       1   #####
       2   #####
       3   #####
       4   #####
       5   #####
       6   #####
       7   #####

59(;):  76543210
       0    #
       1    #
       2    #
       3########
       4########
       5########
       6########
       7########

60(<):  76543210
       0########
       1########
       2########
       3########
       4########
       5    #
       6    #
       7    #

61(=):  76543210
       0#####
       1#####
       2#####
       3#####
       4########
       5#####
       6#####
       7#####

62(>):  76543210
       0   #####
       1   #####
       2   #####
       3   #####
       4########
       5   #####
       6   #####
       7   #####

63(?):  76543210
       0   #####
       1   #####
       2   #####
       3   #####
       4   #####
       5
       6
       7

64(@):  76543210
       0
       1
       2
       3   #####
       4   #####
       5   #####
       6   #####
       7   #####

65(A):  76543210
       0
       1
       2
       3#####
       4#####
       5#####
       6#####
       7#####

66(B):  76543210
       0#####
       1#####
       2#####
       3#####
       4#####
       5
       6
       7

67(C):  76543210
       0   #####
       1   #####
       2   #####
       3   #####
       4   #####
       5  #
       6 #
       7#

68(D):  76543210
       0#
       1 #
       2  #
       3   #####
       4   #####
       5   #####
       6   #####
       7   #####

69(E):  76543210
       0       #
       1      #
       2     #
       3#####
       4#####
       5#####
       6#####
       7#####

70(F):  76543210
       0#####
       1#####
       2#####
       3#####
       4#####
       5     #
       6      #
       7       #

71(G):  76543210
       0       #
       1
       2
       3
       4
       5
       6
       7

72(H):  76543210
       0
       1
       2
       3
       4
       5
       6
       7       #

73(I):  76543210
       0
       1
       2
       3
       4
       5
       6
       7#

74(J):  76543210
       0#
       1
       2
       3
       4
       5
       6
       7

75(K):  76543210
       0########
       1
       2
       3
       4
       5
       6
       7
```

```
76(L):  76543210
       0
       1
       2
       3
       4
       5
       6
       7########

77(M):  76543210
       0#
       1#
       2#
       3#
       4#
       5#
       6#
       7#

78(N):  76543210
       0       #
       1       #
       2       #
       3       #
       4       #
       5       #
       6       #
       7       #

79(O):  76543210
       0
       1########
       2
       3
       4
       5
       6########
       7

80(P):  76543210
       0
       1########
       2#
       3#
       4#
       5#
       6########
       7

81(Q):  76543210
       0
       1########
       2##
       3##
       4##
       5##
       6########
       7

82(R):  76543210
       0
       1########
       2###
       3###
       4###
       5###
       6########
       7

83(S):  76543210
       0
       1########
       2####
       3####
       4####
       5####
       6########
       7

84(T):  76543210
       0
       1########
       2#####
       3#####
       4#####
       5#####
       6########
       7

85(U):  76543210
       0
       1########
       2######
       3######
       4######
       5######
       6########
       7

86(V):  76543210
       0
       1########
       2#######
       3#######
       4#######
       5#######
       6########
       7

87(W):  76543210
       0
       1########
       2########
       3########
       4########
       5########
       6########
       7

88(X):  76543210
       0
       1       #
       2       #
       3       #
       4       #
       5       #
       6       #
       7

89(Y):  76543210
       0
       1#
       2#
       3#
       4#
       5#
       6#
       7

90(Z):  76543210
       0#      #
       1 #    #
       2  #  #
       3   ##
       4   ##
       5  #  #
       6 #    #
       7#      #

91([):  76543210
       0    #
       1    #
       2    #
       3    #
       4########
       5    #
       6    #
       7    #

92(\):  76543210
       0   ##
       1  ####
       2## ## ##
       3   ##
       4   ##
       5   ##
       6   ##
       7

93(]):  76543210
       0   ##
       1   ##
       2   ##
       3   ##
       4## ## ##
       5  ####
       6   ##
       7

94(^):  76543210
       0   ##
       1  ####
       2## ## ##
       3   ##
       4## ## ##
       5  ####
       6   ##
       7

95(_):  76543210
       0########
       1#      #
       2#      #
       3#      #
       4#      #
       5#      #
       6#      #
       7########

96('):  76543210
       0  ####
       1 ##  ##
       2     ##
       3    ##
       4   ##
       5
       6   ##
       7

97(a):  76543210
       0##   #
       1# # #
       2#  #
       3##
       4# #
       5#  #
       6#
       7

98(b):  76543210
       0 ##
       1 # #
       2 #  #
       3 ###
       4 #  #
       5 # #
       6 ##
       7

99(c):  76543210
       0   #
       1   ##
       2   # #
       3#  #  #
       4 # #
       5  ##
       6   #
       7
```

```
100(d):  76543210  101(e):  76543210  102(f):  76543210  103(g):  76543210
        0#     #           0#     #           0#  # #           0#     #
        1##   ##           1##   ##           1# # #           1 #   #
        2# # # #           2# # # #           2## #           2  # #
        3#  #  #           3#  #  #           3# #            3   #
        4# # # #           4#     #           4##             4  # #
        5##   ##           5#     #           5#              5 #   #
        6#     #           6#     #           6#              6#     #
        7                  7                  7               7

104(h):  76543210  105(i):  76543210  106(j):  76543210  107(k):  76543210
        0##    #           0   #           0   #           0   #
        1# #   #           1   #           1  ###          1   #
        2## #  #           2   #           2 # # #         2   #
        3# # # #           3   #           3#  #  #        3  ###
        4#  # ##           4   #           4 # # #         4 # # #
        5#   # #           5   #           5  ###          5#  #  #
        6#    ##           6   #           6   #           6#  #  #
        7                  7                  7               7

108(l):  76543210  109(m):  76543210  110(n):  76543210  111(o):  76543210
        0   #           0##   ##           0#  #           0##   #
        1   ##          1# # # #           1 # #           1# # ##
        2   # #         2#  #  #           2  ###          2## # #
        3   #  #        3# # # #           3   # #         3# # #
        4   #           4##   ##           4   #  #        4#  #
        5   #           5#     #           5   #           5#
        6   #           6#     #           6   #           6#
        7               7                  7               7

112(p):  76543210  113(q):  76543210  114(r):  76543210  115(s):  76543210
        0#          0 #         0 ##          0 #
        1#          1 #         1 # #         1 #   #
        2#          2 #         2 #  #        2 #  ##
        3#  #       3 ####      3 # #         3 # # #
        4# # #      4 #   #     4 ##          4 ##  #
        5##   #     5 #   #     5 # #         5 #   #
        6#     #    6 #   #     6 #  #        6     #
        7           7                  7               7

116(t):  76543210  117(u):  76543210  118(v):  76543210  119(w):  76543210
        0   #           0 ##          0   #           0 ##
        1  ###          1 # #         1# ### #        1 # #
        2 # # #         2 #  #        2 # # #         2 #  #
        3#  #  #        3 #   #       3   #           3 # #
        4   #           4 #   #       4   #           4 ##
        5   #           5 #   #       5   #           5 #
        6   #           6 #   #       6   #           6 #
        7               7                  7               7

120(x):  76543210  121(y):  76543210  122(z):  76543210  123({):  76543210
        0#  #  #           0###                0   #           0###  ###
        1 # # #           1## #              1  # #          1##    ##
        2  ###            2# # #             2 #   #         2  #  #
        3   #             3#  # #            3  # #          3###  ###
        4   #             4#  ## #           4   #           4###  ###
        5   #             5#  # ##           5  # #          5###  ###
        6   #             6#  #  #           6 #   #         6###  ###
        7                 7                  7               7########
```

```
124( ):  76543210  125(}):  76543210  126(~):  76543210
        0###  ###          0###  ###          0##    ##
        1###  ###          1##    ##          1#  ##  #
        2###  ###          2  #  #            2#####  #
        3###  ###          3###  ###          3####  ##
        4  #  #            4  #  #            4###  ###
        5##    ##          5##    ##          5########
        6###  ###          6###  ###          6###  ###
        7########          7########          7########
```

Remarks

Two different versions of font 3 were supplied by Infocom, which we shall call the Amiga and PC forms (the Atari form is the same as for the PC). The arrow shape differed slightly and so did the rune alphabet. Each was an attempt to map the late Anglian ("futhorc") runic alphabet, which has 33 characters, onto our Latin alphabet. The drawings above are from the Amiga set.

Most of the mappings are straightforward (e.g., Latin A maps to Anglian a), except that: Latin C is mapped to Anglian eo; K to "other k" (previously a z sound); Q to Anglian k (the same rune as c); V to ea; X to z and Z to oe. The PC runes differ as follows: G has an ornamental circle making it more look like "other z"; K maps to Anglian k (or c); Q is an Anglian ea (which resembles the late Anglian q); V is an oe; X is an "other k" and Z is a symbol Infocom seem to have invented themselves. (Though less well drawn the PC runes arguably have a better sound-mapping.)

The font behaviour of 'Beyond Zork', which does have bit 3 of 'Flags 2' set, is rather complicated and depends on the interpreter number it finds in the header (see **S** 11). Specifically:

1.(a Digital terminal) 'BZ' asks whether the player has a VT220 terminal (a model capable of character graphics) and uses font 3 if and only if the answer is yes. (An in-house convenience: Infocom used a Digital mainframe.)

2.(Apple IIe) 'BZ' never uses font 3.

3.(Macintosh) 'BZ' always uses font 3.

4.(Amiga) 'BZ' always uses font 3.

5.(Atari ST) 'BZ' always uses font 3.

6.(MSDOS) 'BZ' uses font 3 if it finds bit 3 of 'Flags 2' set (indicating that a graphical screen mode is in use) and otherwise uses IBM PC graphics codes. These need to be converted back into ASCII. The conversion process used by the **Zip** interpreter is as follows:

```
179   becomes   a vertical stroke (ASCII 124)
186             a hash (ASCII 35)
196             a minus sign (ASCII 45)
205             an equals sign (ASCII 61)
all others in the range 179 to 218 become a plus sign (ASCII 43)
```

7.(Commodore 128) 'BZ' always uses font 3.

8.(Commodore 64) 'BZ' always uses font 3.

9.(Apple IIc) 'BZ' uses Apple character graphics (possibly "Mousetext"), but has problems when the units used are not 1×1.

10.(Apple IIgs) 'BZ' always uses font 3.

11.(Tandy) 'BZ' crashes on the public interpreters.

A similarly tangled process is used in 'Journey'. It is obviously highly unsatisfactory to have to make the decision in the above way, which is why **set_font** is now required to return 0 indicating non-availability of a font.

Stefan Jokisch suggests that Infocom originally intended the graphics bit as a way to develop Version 5 to allow a graphical version in parallel with the normal text one. For instance, when the Infocom MSDOS interpreter starts up, it looks at the graphics flag and:

if clear, it sets the font width/height to 1/1 (so that screen units are character positions);

if set, it enters MGCA, a graphical screen mode and sets the font width/height to 8/8 (so that screen units are pixels).

The "COLOR" command in 'BZ' (typed at the keyboard) also behaves differently depending on the interpreter number, which is legal behaviour and has no impact on the specification.

Appendix A. Error messages and debugging

Older interpreters, such as **ITF**, are extremely curt when an error condition is reached (for example, an illegal opcode). It was assumed that Infocom's shipped story files were bug-free, which is mostly true, so that errors could only arise through a bug elsewhere in the interpreter.

In debugging Inform games, though, many error conditions can arise and it is extremely helpful to report these as fully as possible. These include:

1. An illegal opcode being hit;

2. A call to what can't be a routine (because the initial byte is not between 0 and 15);

3. A jump or call to an address beyond the size of the story file;

4. An attempt to **print_obj**, or otherwise access, an object which doesn't exist, such as object number 0.

5. An attempt to write to, or get the property length of, a nonexistent property.

6. An attempt to access an attribute outside the range 0 to 31 or 0 to 47 (depending on Version). (But note that Infocom's 'Sherlock' contains a bug causing it to try setting or clearing attribute number 48.)

7. Division by zero. The player sometimes then has the annoying task of working out where the error took place in source code. Providing a stack back-trace would be a help.

As mentioned in **S** 3, it's helpful to screen out any illegal ZSCII characters between 0 and 31 which are accidentally printed: crashes can be very mysterious when they cause interpreters to send control codes to the terminal.

In addition, an interpreter might provide options for keeping track of maximum stack usage and the typical number of opcodes executed between each read from the keyboard. (But watching these is a time-wasting activity, so they should be options.)

Finally, infinite loops fairly often happen, as in any programming language. On a system without pre-emptive multi-tasking, this may lock up the whole machine, as the usual way that porters implement multi-tasking is to return control to the host operating system only when the keyboard is read. This can be avoided by providing a point in the code which could return control to the OS from time to time (say, every 2000 instructions).

Appendix B. Conventional contents of the header

The header table in section 11 details everything the interpreter needs to know about the header's contents. A few other slots in the header are meaningful but only by convention (since existing Infocom and Inform games write them). These additional slots are described here.

As in **S** 11, "Hex" means the address, in hexadecimal; "V" the earliest version in which the feature appeared; "Dyn" means that the byte or bit may legally be changed by the game during play (no others may legally be changed by the game); "Int" means that the interpreter may (in some cases must) do so.

Conventional usage of unspecified header memory

Hex	V	Dyn	Int	Contents
1	1			*Flags 1:*
	3		*	Bit 3: The legendary "Tandy" bit (see note)
2	1			Release number (word)
10	1	*		*Flags 2:*
	3			Bit 4: Set in the Amiga version of The Lurking Horror so presumably to do with sound effects?
	?	?	*	10: Possibly set by interpreter to indicate an error with the printer during transcription
12	2			Serial code (six characters of ASCII)
	3			Serial number (ASCII for the compilation date in the form YYMMDD)
38	6		*	Infocom used this as 8 bytes of ASCII: the player's user-name on their own mainframe, useful to identify which person played a saved-game (however, the bytes are all 0 in shipped story files)
3C				Inform 6 stores 4 bytes of ASCII here, giving the version of Inform used to compile the file: for instance, "6.11".

1. In Versions 1 to 3, bits 0 and 7 of 'Flags 1' are unused. (The meaning of bit 2 has recently been discovered: see section 11.) In later Versions, bits 0, 6 and 7 are unused. In 'Flags 2', bits 9 and 11-15 are unused. Infocom used up almost the whole header: only the bytes at **$32** and **$33** are unused in any Version, and those are now allocated for standard interpreters to give their Revision numbers.

2. Some early Infocom games were sold by the Tandy Corporation, who seem to have been sensitive souls. 'Zork I' pretends not to have sequels if it finds the Tandy bit set. And to quote Paul David Doherty:

> In 'The Witness', the Tandy Flag can be set while playing the game, by typing **$DB** and then **$TA**. If it is set, some of the prose will be less offensive. For example, "private dicks" become "private eyes", "bastards" are only "idiots", and all references to "slant-eyes" and "necrophilia" are removed.

We live in an age of censorship.

3. For comment on interpreter numbers, see **S** 11. Infocom's own interpreters were generally rewritten for each of versions 3 to 6. For instance, interpreters known to have been shipped with the Macintosh gave version letters B, C, G, I (Version 3), E, H, I (Version 4), A, B, C (Version 5) and finally 6.1 for Version 6. (Version 6 interpreters seem to have version numbers rather than letters.) See the "Infocom fact sheet" for fuller details.

4. Inform 6 story files are easily distinguished from all other story files by their usage of the last four header bytes. Inform 1 to 5 story files are best distinguished from Infocom ones by the serial code date: anything before 930000 is either an Infocom file, or a fake. (The author of 'Jigsaw' is tempted to compile a millenial version with serial code 991231 when the time comes, but then the next day serial codes will clock over to 000101. The decision of how to continue serial codes past the year 2079 is deferred to a future revision of this Standard.) Clearly there is no point going to any trouble to prevent fakes, but with a little practice it's easy to tell whether Zilch or Inform compiled a file from the style of code generated.

Appendix C. Resources available

> ...the dead hand of the academy had yet to stifle the unbridled enthusiasms of a small band of amateurs in Europe and America.
>
> *Michael D. Coe,* Breaking the Maya Code

[Note: the hypertext links in this appendix are to WWW sites external to the standards document, and were correct as of 20th June 1997.]

The resources below are mainly available from the if-archive at the anonymous FTP site **ftp.gmd.de** in Germany, maintained by Volker Blasius. [Some mirrors of this site are listed at the Inform home page.]

Public Interpreters

At least eleven essentially different interpreters are publically available, of which six are in modern use:

1. Frotz, by Stefan Jokisch (1996-), was written in a successful attempt to implement a Standard-compliant interpreter from scratch, rather than repair old interpreter cores. It covers all Versions and ports are available for: DOS, Amiga, Windows CE, Acorn RISC OS, Windows 95/NT, OS/2, Unix, HP-UX.

2a. Zplet, by Matthew Russotto (1996-), is an almost Standard-compliant Java applet to interpret Versions 3, 5 and 8.

2b. Zax, by Matt Kimmel (1997-), is an almost Standard-compliant Java application to interpret all Versions other than 6. (Note that **Zax** and **Zplet** are quite independent of each other: as one runs in a browser, the other as a stand-alone application, they may be seen as complementary.)

3. Zip, by Mark Howell (1991-), is a good, almost entirely accurate interpreter across Versions 1 to 5. The core has evolved into many extended versions, often covering Version 8 or even implementing this Standard; Stefan Jokisch has written a document listing bugs and omissions in the original. Noteworthy ports include: Kevin Bracey's Standard-compliant !Zip2000, (Acorn RISC OS only), which fully supports even Version 6; Rick Bram's Pilot Zip (1997): the PalmPilot is a pocket-sized, battery-powered personal organiser without even a keyboard; Andrew Plotkin's **MaxZip**, for System 7 Macintoshes, and **XZip**, for X-Windows; Matthew Russotto's **Zip Infinity**, also for Macintosh; John Holder's **JZip** for PCs (itself ported to the Atari ST and Bebox); Greg Ewing's **macZeX**, which extends the Z-machine specification to include textual formatting information loosely based on TeX, but which has not been used by designers; and other ports including to MS Windows, DOS, BSD Unix, Amiga, OS/2, Apple IIgs.

4. The InfoTaskForce (or ITF) interpreter (1987-92) is almost as good, but slower and less accurate on some Version 5 features. It is no longer maintained by its original authors (David Beazley, George Janczuk, Peter Lisle, Russell Hoare and Chris Tham) and the final version was 4.01 (ported to Acorn RISC OS, Atari ST, OS/2, Macintosh, DOS, Amiga); a beta test of version 4.02 was never widely distributed. However, Bryan Scattergood has given **ITF** a new lease of life by updating it to much more accurate and reliable interpreters for Acorn RISC OS, the Psion

Series 3, Unix/X11 and Windows.

5. Apple Newton users can at present play Version 3 games only, using the shareware interpreter YAZI, Yet Another Z-Machine Interpreter, by George Madrid and Sanjay Vankil (1994-).

6. Another Version-3-only interpreter, Infocom Interpreter by Martin Korth (1993) is noteworthy for two reasons: first, because Mr Korth seems to have worked by reverse-engineering the Infocom CP/M interpreter (in isolation from the main groups of Infocom hackers of the period), and in this way wrote the only known Z-machine in Pascal (source available at his site); second, because he then wrote an assembly-language version for the keyboard-less Nintendo Gameboy. To use this, one appends the story file to the interpreter, burns the result into an EPROM and plugs it in: it's probably the nearest thing to a "hardware Z-machine" yet devised.

The other four interpreters are obsolescent and now hardly used, but ought not to be forgotten, if only for their contribution to the gradual process of decipherment.

7. Pinfocom (1992), derived from an early form of **ITF**, and released by Paul Smith as a Version 3 (only) interpreter; final version 3.0 (ported to Amiga and Atari ST).

8. Zmachine (1988-90), by Matthias Pfaller: briefly in limited circulation (again, for Version 3 only; ported to Amiga and Atari ST).

9. ZIPDebug (1991-3), by Frank Lancaster, supporting Versions 1 to 5 and offering some debugger facilities.

10. Zterp (1992), by Charles M. Hannum, for Versions 3 to 5: reputedly very fast.

Testing compliance

Andrew Plotkin has written a story file to torture interpreters into revealing non-Standard behaviour, with the appropriately contrived name of TerpEtude [an archive containing the source code and compiled story file]. It supersedes the handful of smaller programs previously attached to versions of this document.

Compilers

Infocom's original compiler **Zilch** no longer exists: nor is any of its language, **ZIL**, documented anywhere (though this is similar to **MDL**, which is documented): no continuous part of the source code of any of Infocom's games is in the public domain [but see Stu Galley's chapter of an Infocom history article, and the IEEE article, for fragments].

Inform is the only other compiler to have existed. It is freeware and comes with full documentation (of which this document is a part).

Debugger

A source-level debugger for Inform games, called **Infix**, has been on the drawing boards for some years now. A group of authors is currently developing an implementation.

Utility programs

Mark Howell has written a toolkit of Ztools, or utility programs (1991-5, updated 1997), which includes:

1. Txd, a disassembler for Versions 1 to 8. (Uses the same opcode names as Inform and this document, and has an option to disassemble in Inform assembly-language syntax.)

2. Infodump, capable of printing the header information, object tree (with properties and attributes), dictionary and grammar tables of any Infocom or Inform-compiled game. (Understands all four varieties of grammar table: Infocom pre-Version 6, Infocom Version 6, Inform GV1 and GV2.)

3. Pix2gif, for converting Version 6 picture data to GIF files.

4. Check, for verifying Infocom or Inform story files.

These continue to be maintained (by Matthew Russotto) and the first two are extremely useful. **Infodump** largely supersedes Mike Threepoint's vocabulary dumper **Zorkword** (1991-2), which was important in its day (and which this author found extremely helpful when writing Inform 1).

Story files

1. Numerous Inform-compiled story files are publically available: games such as 'Curses', 'Christminster', 'Theatre', 'Busted', 'Balances', 'Advent', 'Adventureland' and so on. [For an annotated selection see the Inform home page.]

2. A few Infocom story files are public, notably two 4-in-1 sample games (released for advertising purposes: 55.850823 and 97.870601) and Minizork (a heavily abbreviated form of Zork I released with a Commodore magazine).

3. Almost all Infocom's games remain commercially available in anthologies published by Activision. Copyright resides in them and they should not available by FTP from any site.

4. A few other Infocom story files have existed but are neither on sale nor released from copyright: this applies to several of the Version 6 games, those games involving literary rights or other legal issues ('Shogun', 'Hitch-Hiker's Guide To The Galaxy') and ephemera such as beta-test versions (notably the German version of 'Zork I') which have somehow passed into private circulation.

Most of the Infocom games exist in several different releases, and some were written for one Version and then ported to later ones. 'Zork I', for instance, has at least 11 releases, 2 early, 8 in Version 3 (with release numbers between 5 to 88 in chronological order) and one in Version 5 (release 52 -- the releases go back to 1 when the version changes).

Version 1 and 2 games are extinct, though there are a few fossils in the hands of collectors.

Documents

The definitive guide to all Infocom story files known to exist, and an indispensable reference for anyone interested in Infocom, is Paul David Doherty's Infocom fact sheet, which is regularly updated, concise and precise. This supersedes Paul Smith's "Infocom Game Information" file.

Stefan Jokisch has written a brief specification of Infocom-format sound effects files.

Martin Frost is the author of the Quetzal standard for saved-game files. Patches to adapt **Zip**-based interpreters to use **Quetzal** are now available.

Andrew Plotkin is drafting the **Blorb** standard for packaging up images and sounds with new Z-machine games.

The Inform Technical Manual documents the format of parsing tables used in Inform games.

Most of the contents of the original Infocom game manuals are still on sale with the games themselves: the "samplers" (sample transcripts of play) are not, but an archive of them is publically available. So is an interesting historical archive of magazine articles concerning Infocom, and articles from Infocom's own publicity magazine [indexed here].

Mailing list

A Z-Machine mailing list, organised by Marnix Klooster (marnix@worldonline.nl), enables debates on this document, discussion of what interpreters should do, collaboration on new programs and so on.

Appendix D. A short history of the Z-machine

Infocom made six main Versions of the Z-machine and several minor variant forms. These are recognisably similar but with labyrinthine differences, like different archaic dialects of the same language. (The archaeological record stops sharply in 1989 when the civilisation in question collapsed.)

Broadly, these fall into two groups: early (Versions 1 to 3) and late (4 to 6). More fully:

```
Version 1   Early Apple II and TRS-80 Model I games

Version 2   Early Apple II and TRS-80 Model I games

Version 3   "Standard" series games

Version 4   "Plus" series games

Version 5   "Advanced" series games, or, as the marketing division would
            have it, "Solid Gold Interactive Fiction" -- a reference to
            the colour (though not composition) of the boxes they came in

Version 6   Later games with graphics, mouse support, etc.
```

Infocom called their own interpreters ZIP (versions 1 to 3), EZIP/LZIP (V4), XZIP (V5) and YZIP (V6). They speculated on the possibility of an interpreter capable of running all Versions, but never published one.

The original purpose of the Z-machine was simply to implement as much as possible of the mainframe game "Zork" on the first popular wave of home computers.

(Apparently "zork" was a nonsense word used at MIT for the current uninstalled program in progress, and stuck. Just as this document uses the term "Z-machine" for both the machine and its loaded program (which is also sometimes called the "story file"), so ZIP (Zork Implementation Program) was used to mean either the interpreter or the object code it interpreted. Code was written in ZIL (Zork Implementation Language), which was derived from MDL (informally called "muddle"), a particularly unhelpful form of LISP. It was then compiled by ZILCH to assembly code which was passed to ZAP to make the ZIP.)

The Z-machine as originally constructed was surprisingly similar to that in use today. Version 1 (by Joel Berez and Marc Blank, in Autumn 1979) contained essentially all of the main architecture: the header, the memory divided into three, the variables and stack, the object tree, the dictionary, the instruction format. It used "shift lock" characters (a text compression trick which did not survive, though it was more efficient on long sequences of capital letters or punctuation characters than the technique which replaced it). The first micro interpreters were for the TRS-80 Model I (by Scott Cutler) and the Apple II (by Bruce K. Daniels). (A TRS-80 Model II interpreter was written but never actually shipped.)

Version 2 was only a minor enhancement. Abbreviations (used to help text compression) appeared, but only in one 32-word bank, and the six-digit serial number appeared in the header, though it wasn't always the date in those days: Release 7 of 'Zork II', for instance, is numbered **UG3AU5**. (Other bizarre serial numbers, such as **000000**, appear on fakes or beta-test releases.)

In Version 3, the text encoding alphabets changed again, and the old "shift lock" codes were dropped in favour of expanding the abbreviations bank to 96 entries. The "verify" opcode and checksums appeared; and a new opcode to reprint the status line at the top of the screen was introduced. (Previously, this had been updated only when input was taken from the keyboard.) The earliest Version 3 releases ('Deadline', then reissues of 'Zork I' and 'II') were in March and April 1982; the last (the 'Minizork', a cassette-based Commodore-64 sample of 'Zork') was in Novem-

ber 1987.

The idea of widespread portability finally came of age as (between 1982 and 1985) interpreters were developed for the Atari 400/800, CP/M, the IBM PC, the TRS-80 Model III, the NEC APC, the DEC Rainbow, the Commodore 64, the TI Professional, the DECmate, the Tandy-2000, the Kaypro II, the Osborne 1, MS-DOS, the TI 99/4a, the Apple Macintosh, the Epson QX-10, the Apricot, the Atari ST and the Amiga. Infocom's middle period coincided with the bubble in home computers, before the market collapsed to its present apparently stable state (in which IBM and Apple share almost the entire market), and the Z-machine's portability gave Infocom a unique advantage over its competitors. Also, it was an expertly marketed quality brand at a time when standards of workmanship were very variable; and text-only games did not seem so dull at a time when graphics were on the whole crude and slow. These factors combined to give Infocom considerable (though never enormous) commercial success.

By 1982, then, the Z-machine had stabilised to a clean design which was to remain in use for six years. It was very portable, contained everything reasonably necessary and most of its complications were badly-needed space optimisations. (Although Version 3 can fit 128K of story file, the practical limit in 1982-4 was about 110K, that being the typical disc capacity on target machines.) The ZAP assembler was cleverly written to exploit these optimisations, though the Zilch compiler's code generator was much less efficient. (Interestingly, Infocom did not develop any generic central library, and Infocom's authors worked fairly independently of each other: each new game would inherit a small core of code from a previous one, but this would make up only about 10K of code (about a third of the size of the Inform library) and would end up being hacked about to suit the new game. Without a central library, Infocom games waste a fair amount of space in duplicating code for routine operations many times over. For this reason, Inform games tend to squash appreciably more design into the format.)

"Verify" and checksum data were quickly introduced. However, the first serious variant on Version 3 was made in 1984 when a primitive form of screen-splitting was invented to give 'Seastalker' a sonar display. This design (perhaps accidentally) became the foundation for the graphics systems of later versions.

Much later (in 1987) sound effects were added to Version 3 for 'The Lurking Horror', though by that time it was really a Version 5 feature being passed down to the old model (and only to the Amiga interpreter in any case). ('TLH' is contemporaneous with 'Sherlock' (in Version 5), the only other game to actually use the sound effects features.)

During 1983-5, Infocom poured resources into an ambitious pet project of its founders: 'Cornerstone', a database which used some of the same portable virtual machine ideas as the Z-machine. The business market, however, was not nearly as diverse as the home computer market: 'Cornerstone' probably was the best database available on the Atari ST, but it made no impression on the IBM PC market. The result was a commercial failure which compounded the company's over-expansion problems (driving it into a merger with Activision), though it certainly did not destroy Infocom's viability.

By 1985, Infocom had begun to write interpreters in C for the sake of portability (previously, a different assembly-language program had to be maintained for every model of computer). The main motivation to keep the format stable was therefore largely removed: it became possible to upgrade the Z-machine for every new game, if need be.

There were two basic pressures for change. One was that home computers were larger, and several fundamental restrictions (the game size being only 128K, the number of objects only 255, the attributes only 32, the properties only 31) were beginning to bite. The other was the drive for more gimmicks - character graphics, flashier status lines, sound effects, different typefaces, and so on. The former led to logical, easy to understand structural changes in the machine (designed

by Marc Blank). The latter, in contrast, made a mess of the system of opcodes (designed by committee).

More does not mean better (halving the price of paper does not double the quality of the novel). The relieving of size restrictions only increased design time -- or endangered the quality of the designs being produced. The Version 3 games have a spare, concise literary style which is absent from the later games. (But Inform authors have certainly found Version 3 slightly too small for comfort, and it's useful to be able to spill over its boundaries.)

In August the first Version 4 game ('A Mind Forever Voyaging') reached production. Opinions vary as to whether it was brilliant or awful, but it was certainly a departure (and could not have been written under Version 3). In retrospect there is no doubt about 'Trinity', now generally considered the finest game written: it had previously been shelved as too ambitious for the Version 3 format. Still, most of the new 1985/6 games remained in Version 3: there were still plenty of 8-bit home computers around which were too small for Version 4 games. Despite critical acclaim, the new games consequently did not sell as well. (Brian Moriarty commented that 'Trinity' "sold tolerably well. Better than we'd hoped." But his previous game, the more modest 'Wishbringer', had sold rather better.)

Version 5 games began to appear in September 1987 with 'Beyond Zork' and 'Border Zone'. Both of these games needed new features -- character graphics run wild in the case of the former, and real-time keyboard interaction in the latter. The number of opcodes grew ever faster as a result.

Although five old games were re-released in Version 5 editions (with an in-game hints system added, and benefiting from 9-letter word dictionaries, but otherwise as written), the direction was all too clearly away from the old text game into graphics: 'Beyond Zork' can look like a parody of an early mainframe maze game, for instance. Version 6 completed the process during something of a hiatus in 1988, after which the last few increasingly-unrecognisable Infocom games appeared: 'Zork Zero', 'Shogun', 'Journey' and 'Arthur'.

It would be wrong, though, to suggest that Infocom regarded text and graphics as incompatible opposites. Infocom had never been puritanically opposed to graphics --

> We have nothing against graphics per se. However, given the quality of graphics currently available on home computers, we would rather use that disk space for additional puzzles and richer descriptions.
>
> **The New Zork Times (Spring 1984)**

(and, after all, the same author wrote both 'Trinity' and 'Beyond Zork'). Although the old Infocom parser was considered to have passed its sell-by date, Version 6 did not drop textual input in favour of some inane point-and-click interface. Instead, an entirely new parser was devised from scratch ("using the theory of computational linguistics", according to a puff by Stu Galley: broadly an LALR(1) parser).

Infocom gradually ceased to exist during 1987-9 as its financial problems grew. But its products were increasingly regarded as anachronistic and most of its staff had left since the middle years: if Infocom had not finally been wound up, whether it would have continued to release text games of the classical style is arguable.

Two new formats, versions 7 and 8, have recently been devised to cope with large Inform games.

Appendix E. Statistics

> LORD DIMWIT FLATHEAD: "It must have two hundred thousand rooms, four million takeable objects, and understand a vocabulary of every single word ever spoken in every language ever invented."
>
> *The New Zork Times (Winter 1984)*

To give some idea of the sizes found in typical story files, here are a few statistics, mostly gathered by Paul David Doherty, whose "Infocom fact sheet" file is the definitive reference.

(i) *Length*

The shortest files are those dating from the time of the 'Zork' trilogy, at about 85K; middle-period Version 3 games are typically 105K, and only the latest use the full memory map. In Versions 4 and 5, only 'Trinity', 'A Mind Forever Voyaging' and 'Beyond Zork' use the full 256K. 'Border Zone' and 'Sherlock', for instance, are about 180K. (The author's short story 'Balances' is about 50K, an edition of 'Adventure' takes 80K, and 'Curses' takes 256K (it's padded out to the maximum size with background information; the actual game comprises only about 245K). Under Inform, the library occupies about 35K regardless of the size of game.)

(ii) *Code size*

'Zork I' uses only about 5500 opcodes, but the number rises steeply with later games; 'Hollywood Hijinx' has 10355 and, e.g. 'Moonmist' has 15900 (both these being Version 3). Against this, 'A Mind Forever Voyaging' has only 18700, and only 'Trinity' and 'Beyond Zork' reach 32000 or so. (Inform games are more efficiently compiled and make better use of common code -- the library -- so perform much better here: the old Version 3, release 10 of 'Curses' (128K long, and a larger game than any Infocom Version 3 game) has only 6720 opcodes.)

(iii) *Objects and rooms*

This varies greatly with the style of game. 'Zork I' has 110 rooms and 60 takeable objects, but several quite complex games have as few as 30 rooms (the mysteries, or 'Hitch-hikers'). The average for Version 3 games is 69 rooms, 39 takeable objects.

'A Mind Forever Voyaging' contains many rooms (178) but few objects (30). 'Trinity', a more typical style of game, contains 134 rooms and 49 objects: the Version 5 'Curses' has a few more of each. Of the Version 6 games, only 'Zork Zero' scores highly here, with 215 rooms and 106 objects. The average for Version 4/5 games is 105 rooms and 54 objects.

The total number of objects tends to be close to the limit of 255 in Version 3 games. 'Curses' contains 508.

(iv) *Dictionary*

Early games such as 'Zork I' know about 600 words, but again this rises steeply to about 1000 even in Version 3. Later games know 1569 ('Beyond Zork') to the record, 2120 ('Trinity'). (This is achieved by heroic inclusion of unlikely synonyms: e.g. the Japanese lady with the umbrella

can be called WOMAN, LADY, CRONE, MADAM, MADAME, MATRON, DAME or FACE with any of the adjectives OLD, AGED, ANCIENT, JAP, JAPANESE, ORIENTAL or YELLOW.) Version 6 games have smaller dictionaries. So has 'Curses', at 1364.

(v) *Opcodes*

(a) Of the 1426854 opcodes in the shipped Infocom story files in Paul David Doherty's collection, here are the top and bottom ten most popular. (Leaving out those which never occur and so score 0: **nop**, **art_shift**, **piracy** and the two post-Infocom opcodes, **print_unicode** and **check_unicode**.)

Top Ten Opcodes Chart			**Bottom Ten Opcodes Chart**	
1.	je	195959	1.	print_form 2
2.	print	142755	2.	erase_picture 3
3.	jz	112016	3.	read_mouse 3
4.	call_vs	104075	4.	encode_text 7
5.	print_ret	80870	5.	make_menu 9
6.	store	71128	6.	not 14
7.	rtrue	66125	7.	scroll_window 16
8.	jump	56534	8.	pop_stack 17
9.	new_line	52553	9.	restore_undo 18
10.	test_attr	46627	10.	mouse_window 22

So about 2/3rds of all opcodes are those in the top ten; 1 in 8 opcodes is a **je**, and only 1 in 710000 is a **print_form**.

(b) An experiment (conducted with the help of Kevin Bracey) sheds some light on the opcodes most frequently interpreted in typical play. Two very different games ('Zork I', Version 5 "solid gold" edition; 'Museum of Inform', a complex Inform example game) were played for about 50000 cycles of the Z-machine (about 20 moves in 'Zork I', rather less in the 'Museum'). The following table records all opcodes with a frequency of at least 1% (i.e., 0.01):

Zork I Solid Gold (Infocom)		**Museum of Inform (Inform)**	
0.116110	loadb	0.104952	je
0.103990	storeb	0.101151	jz
0.101616	jz	0.092727	jump
0.074979	dec_chk	0.080985	jg
0.066375	add	0.079039	jl
0.066283	je	0.070550	inc
0.060760	store	0.070139	store
0.053867	loadw	0.047058	loadw
0.038095	storew	0.034137	get_prop_addr
0.036428	mul	0.024105	jin
0.032069	inc_chk	0.022734	rtrue
0.030243	jump	0.021583	storew
0.029170	test_attr	0.020075	add
0.020634	call_vs	0.018485	call_vs
0.011184	get_sibling	0.016731	and
		0.016082	loadb
		0.012061	call_vn
		0.011879	test_attr
		0.011824	dec
		0.011687	ret

Adventure games spend most of the time parsing, and the differences between these tables reflect different parser designs (byte arrays versus word arrays and arrays stored in properties) as well as different compiler code generators (Inform does not use **inc_chk** or **dec_chk**, so it uses **inc**, **dec**, **jl** and **jg** correspondingly more). In the case of 'Zork I', about a third of all opcodes are branches: in the case of 'Museum', almost half.

Appendix F. Canonical Story Files

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. The notation

Release number.Serial code

identifies particular story files: for example the first production copy of 'Enchanter' is 10.830810.

Paul David Doherty's investigations into Infocom's released games have resulted in the following list of all known story files compiled by **Zilch**, the Infocom compiler:

Version 1 story file

Zork I	5. (no serial code)

Version 2 story files

Zork I	15.UG3AU5
Zork II	7.UG3AU5

Version 3 story files

Ballyhoo	97.851218
Cutthroats	23.840809
Deadline	18.820311, 19.820427, 21.820512, 26.821108, 27.831005
Enchanter	10.830810, 15.831107, 16.831118, 24.851118, 29.860820
Four-In-One Sampler I	26.840731, 53.850407, 55.850823
Four-In-One Sampler II	97.870601
The Hitch Hiker's Guide To The Galaxy	47.840914, 56.841221, 58.851002, 59.851108
Hollywood Hijinx	37.861215
Infidel	22.830916
Leather Goddesses of Phobos	118.860325? (beta), 50.860711?, 59.860730, 59.861114
Lurking Horror	203.870506, 219.870912 (s), 221.870918 (s)
Mini-Zork I	34.871124
Moonmist	4.860918, 9.861022
Planetfall	20.830708, 29.840118, 37.851003
Plundered Hearts	26.870730

Seastalker	86.840320 (beta), 15.840501, 15.840522, 16.850515, 16.850603
Sorcerer	67.000000? (beta), 4.840131, 6.840508, 13.851021, 15.851108, 18.860904
Spellbreaker	63.850916, 87.860904
Starcross	15.820901, 17.821021
Stationfall	107.870430
Suspect	14.841005
Suspended	5.830222, 7.830419, 8.830521, 8.840521
Wishbringer	68.850501, 69.850920
Witness	13.830524, 18.830910, 20.831119, 21.831208, 22.840924
Zork I	23.820428, 25.820515, 26.820803, 28.821013, 30.830330, 75.830929, 76.840509, 88.840726
Zork II	17.820427, 18.820512, 18.820517, 19.820721, 22.830331, 23.830411, 48.840904 ?.841220?
Zork III	10.820818, 15.830331, 16.830410, 15.840518, 17.840727

Note that the two samplers and the mini-Zork are in the public domain and may be downloaded from Internet archive sites. One form of 'Zork I' can be downloaded freely from Activision's Web pages promoting the Zork brand name.

The two problem children here are 'Seastalker', a submarine game which produces a sonar display across the top of the screen (and thus needs more sophisticated screen control features than the other Version 3 games) and 'The Lurking Horror', which uses sound effects (hence the "(s)" notation).

Version 4 story files

A Mind Forever Voyaging	77.850814, 79.851122
Bureaucracy	86.870212, 116.870602
Nord and Bert Couldn't Make Head Nor Tail Of It	19.870722
Trinity	11.860509, 12.860926

Version 5 story files

Beyond Zork	47.870915, 49.870917, 51.870923, 57.871221
Border Zone	9.871008
The Hitch Hiker's Guide To The Galaxy SG	31.871119
Leather Goddesses of Phobos SG	4.880405
Planetfall SG	10.880531
Sherlock	21.871214, 26.880127 (s)
Wishbringer SG	23.880706

Zork I SG	52.871125
Zork I German	3.880113 (beta)

The "SG" games were "solid gold" revisions of existing Version 3 games, adding on-line hints and an UNDO command. Regrettably these are not the versions distributed by Activision on their recent re-releases of the Infocom back catalogue.

One form of 'Sherlock' uses sound effects. 'Border Zone' introduces timed input. 'Beyond Zork' features a character-graphics font. But the most interesting file is the German translation of 'Zork I', which was never commercially released, introducing an alphabet table to the format.

Version 6 story files

Game	Mac	Amiga	Apple II	IBM
Arthur	54.890606	same as Mac	63.890622?	74.890714
Journey	26.890316	30.890322	77.890616?	83.890706
Shogun	292.890314	295.890321	311.890510?	322.890706
Zork Zero	296.881019	366.890323	383.890602?	393.890714

The rule that story files should be independent of their target computers was dropped for Version 6 games and this leads to copious footnotes and exceptions in sections 8.8 and 16 of the standard. Story files for a particular game are substantially similar to each other but use fonts, pictures and so on slightly differently.

Note that a new Infocom game, *Zork: The Undiscovered Underground* was published by Activision in 1997. 16.970828 is the only public version I know of: however, note that this file was compiled by Inform and not by Zilch. It is therefore not useful as a witness to Z-machine rules.

Z-machine Common Save-File Format Standard

also called Quetzal:

Quetzal Unifies Efficiently The Z-Machine Archive Language

version 1.3b (29th September 1997), by Martin Frost

1. Conventions

1.1

A **byte** is an 8-bit unsigned quantity.

1.2

A **word** is a 16-bit unsigned quantity.

1.3

Bitfields are represented as blocks of characters, with the first character representing the most significant bit of the byte in question. Multi-bit subfields are indicated by using the same character multiple times, and values of 0 or 1 indicate that these bits are always of the specified value. Therefore a bitfield described as **010abbcc cccdd111** would be a two-byte bitfield containing four subfields, **a**, of 1 bit, **b**, 2 bits, **c**, 5 bits, and **d**, 2 bits, together with a field 'hard-wired' to 010 and one to 111.

1.4

All multi-byte numbers are stored in big-endian form: most significant byte first, then in strictly descending order of significance.

1.5

The reader is assumed to already be familiar with the Z-machine; in particular its instruction set, memory map and stack conventions.

1.6

When form type names, which are four characters long, are set in running text, they're set in bold-face with spaces replaced by underscores. Thus **____** means "four spaces" and **(c)_** means three letters of the copyright notation followed by a space.

2. Overall structure

2.1

For the purposes of flexibility, the overall format will be a new IFF type. A standard core is defined, and customised information can be stored by specific interpreters in such a way that it can be easily read by others. The FORM type is **IFZS**.

2.2

Several chunks are defined within this document to appear in the IFZS FORM:

```
'IFhd'          5.4
'CMem'          3.7
'UMem'          3.8
'Stks'          4.10
'IntD'          7.8
```

2.3

Several chunks may also appear by convention in any IFF FORM:

```
'AUTH'          7.2, 7.3
'(c) '          7.2, 7.4
'ANNO'          7.2, 7.5
```

3. Content of dynamic memory

3.1

Since the contents of dynamic memory may be anything up to 65534 bytes, it is desirable to have some form of compression available as an option. Bryan Scattergood's port of ITF uses a method that is both elegant and effective, and this is the method adopted.

3.2

The data is compressed by exclusive-oring the current contents of dynamic memory with the original (from the original story file). The result is then compressed with a simple run-length scheme: a non-zero byte in the output represents the byte itself, but a zero byte is followed by a length byte, and the pair represent a block of **n+1** zero bytes, where **n** is the value of the length byte.

3.3

It is not necessary to compress optimally, if to do so would be difficult. For example, an interpreter that does not store the whole of dynamic memory in physical memory may compress a single page at a time, ignoring the possibility of a run crossing a page boundary; this case can be encoded as two adjacent runs of bytes. It is required, however, that interpreters read encoded data even if it does not happen to be compressed to their particular page-boundary preferences. This is not difficult, requiring merely the maintenance of a small amount of state (namely the

current run length, if any) across page boundaries on a read.

3.4

If the decoded data is shorter than the length of dynamic memory, then the missing section is assumed to be a run of zeroes (and hence equal to the original contents of that part of dynamic memory). This permits the removal of redundant runs at the end of the encoded block; again it is not necessary to implement this on writes, but it must be understood on reads.

3.5

Two error cases are possible on reads: the decoded data may be larger than dynamic memory, and the encoded data may finish with an incomplete run (a zero byte without a length byte). These should be dealt with in whatever way seems appropriate to the interpreter writer.

3.6

Dissenting voices have suggested that compression is unnecessary in today's world of cheap storage, and so the format also includes the capability to dump the contents of dynamic memory without modification. The ability to write such files is optional; the ability to read both types is necessary. It is an error for this dump to be shorter or longer than the expected length of dynamic memory.

3.7

The IFF chunk used to contain the compressed data has type **CMem**. Its format is as follows:

3.7.1

```
4 bytes         'CMem'          chunk ID
```

3.7.2

```
4 bytes         n               chunk length
```

3.7.3

```
n bytes         ...             compressed data as above
```

3.8

The chunk used to contain the uncompressed data has type **UMem**. It has the format:

3.8.1

```
4 bytes         'UMem'          chunk ID
```

3.8.2

```
4 bytes         n               chunk length
```

3.8.3

```
n bytes         ...             simple dump of dynamic memory
```

4. Content of stacks

4.1

One of the biggest differences between current interpreters is how they handle the Z-machine's stacks. Conceptually, there are two, but many interpreters store both in the same array. This format stores both in the same IFF chunk, which has chunk ID **Stks**.

4.2

The IFF format includes a length field on each chunk, so we can write only the used portion of the stacks, to save space. The least recent frames on the stacks are saved first, to ensure that the missing part appears at the end of the data in the file.

4.3

Each frame has the format:

4.3.1

3 bytes	...	return PC (byte address)

4.3.2

1 byte	000pvvvv	flags

4.3.3

1 byte	...	variable number to store result

4.3.4

1 byte	0gfedcba	arguments supplied

4.3.5

1 word	n	number of words of evaluation stack used by this call

4.3.6

v words	...	local variables

4.3.7

n words	...	evaluation stack for this call

4.4

The return PC is a byte offset from the start of the story file.

4.6

The p flag is set on calls made by **call_xN** (discard result), in which case the variable number is meaningless (and should be written as a zero).

4.7

Assigning each of the possible 7 supplied arguments a letter a-g in order, each bit is set if its respective argument is supplied. The evaluation stack count allows the reconstruction of the chain of frame pointers for all possible stack models. Words on the evaluation stack are also stored least recent first.

4.8

Although some interpreters may impose an arbitrary limit on the size of the stacks (such as ZIP's 1024-word total stack size), others may not, or may set larger limits. This means that the size of a stack dump may be larger than will fit. If you cannot dynamically resize your stack you must trap this as an error.

4.9

The stack pointer itself is not stored anywhere in the save file, except implicitly, as the top frame on the stack will be the last saved.

4.10

The chunk itself is simply a sequence of frames as above:

4.10.1

```
4 bytes         'Stks'          chunk ID
```

4.10.2

```
4 bytes         n               chunk length
```

4.10.3

```
n bytes         ...             frames (oldest first)
```

4.11

In Z-machine versions other than V6 execution starts at an address rather than at a routine, and therefore data can be pushed on the evaluation stack without anything being on the call stack. Therefore, in all versions other than V6 a dummy stack frame must be stored as the first in the file (the oldest chunk).

4.11.1

The dummy frame has all fields set to zero except **n**, the amount of evaluation stack used. Note that this may also be zero if the game does not use any evaluation stack at the top level.

4.11.2

This frame must be written even if no evaluation stack is used at the top level, and therefore interpreters may assume its presence on savefiles for V1-5 and V7-8 games.

5. Associated story file

5.1

We now come to one of the most difficult (yet most important) parts of the format: how to find the story file associated with this save file, or the related (but easier) problem of checking whether a given save file belongs to a given story.

5.2

Considering the easier second problem first, the actual name of the story file is often not much use. Firstly, filenames are highly dependent on the operating system in use, and secondly, many original Infocom story files were called simply 'story.data' or similar.

5.3

The method most existing interpreters use is to compare the variables at offsets $2, $12, and $1C in the header (that is, the release number, the serial number and the checksum), and refuse to load if they differ. These variables are duplicated in the file (since the header will be compressed with the rest of dynamic memory).

5.4

This data will be stored in a chunk of type **IFhd**. This chunk must come before the **[CU]Mem** and **Stks** chunks to save interpreters the trouble of decoding these only to find that the wrong story file is loaded. The format is:

5.4.1

```
4 bytes         'IFhd'          chunk ID
```

5.4.2

```
4 bytes         13              chunk length
```

5.4.3

```
1 word          ...             release number ($2 in header)
```

5.4.4

```
6 bytes         ...             serial number ($12 in header)
```

5.4.5

```
1 word          ...             checksum ($1C in header)
```

5.4.6

```
3 bytes         ...             initial PC on restore
```

5.5

If the save file belongs to an old game that does not have a checksum, it should be calculated in the normal way from the original story file when saving. It is possible that a future version of this format may have a larger **IFhd** chunk, but the first 13 bytes will always contain this data, and if the other chunks described herein are present they will be guaranteed to contain the data specified.

5.6

The first problem (of trying to find a story file given only a save file) cannot really be solved in an operating-system independent manner, and so there is provision for OS-dependent chunks to handle this.

5.7

It should be noted that the current state of the **IFhd** chunk means it has odd length (13 bytes). It should, of course, be written with a pad byte (as mentioned in 8.4.1).

6. Miscellaneous

6.1

It must be specified exactly what the magic cookie returned by **catch** is, since this value can be stored in any random variable, on the evaluation stack, or indeed anywhere in memory.

6.2

For greatest independence of internal interpreter implementation, **catch** is hereby specified to return the number of frames currently on the system stack. This makes **throw** slightly inefficient on many interpreters (a current frame count can be maintained internally to avoid problems with **catch**), but this is unavoidable without using two stacks and a fixed-size activation record (always 15 local variables). Since most applications of **catch**/**throw** do not unwind enormous depths, (and they are somewhat infrequent), this should not be too much of a problem.

6.3

The numbers of pictures and sounds do not need specification, since they are requested by number by the story file itself.

7. Extensions to the format

7.1

One of the advantages of the IFF standard is that extra chunks can be added to the format to extend it in various ways. For example, there are three standard chunk types defined, namely **AUTH**, **(c)_**, and **ANNO**.

7.2

AUTH, **(c)_**, and **ANNO** chunks all contain simple ASCII text (all characters in the range 0x20 to 0x7E).

7.2.1

The only indication of the length of this text is the chunk length (there is no zero byte termination as in C, for example).

7.2.2

The IFF standard suggests a maximum of 256 characters in this text as it may be displayed to the user upon reading, although it could get longer if required.

7.3

The **AUTH** chunk, if present, contains the name of the author or creator of the file. This could be a login name on multi-user systems, for example. There should only be one such chunk per file.

7.4

The **(c)_** chunk contains the copyright message (date and holder, without the actual copyright symbol). This is unlikely to be useful on save files. There should only be one such chunk per file.

7.5

The **ANNO** chunk contains any textual annotation that the user or writing program sees fit to include. For save files, interpreters could prompt the user for an annotation when saving, and could write an **ANNO** with the score and time for V3 games, or a chunk containing the name/version of the interpreter saving it, and many other things.

7.6

The **ANNO**, **(c)_** and **AUTH** chunks are all user-level information. Interpreters must not rely on the presence or absence of these chunks, and should not store any internal magic that would not make sense to a user in them.

7.7

These chunks should be either ignored or (optionally) displayed to the user. **(c)_** chunks should be prefixed with a copyright symbol if displayed.

7.8

The save-file may contain interpreter-dependent information. This is stored in an **IntD** chunk, which has format:

7.8.1

```
4 bytes         'IntD'          chunk ID
```

7.8.2

```
4 bytes         n               chunk length
```

7.8.3

```
4 bytes         ...             operating system ID
```

7.8.4

```
1 byte          000000sc        flags
```

7.8.5

```
1 byte          ...             contents ID
```

7.8.6

```
2 bytes         0               reserved
```

7.8.7

```
4 bytes         ...             interpreter ID
```

7.8.8

```
n-12 bytes      ...             data
```

7.9

The operating system and interpreter IDs are normal IFF 4-character IDs in form. Please register IDs used with Martin Frost (at the email address given below) so that this can be managed sensibly. They can then be added to future versions of this specification, and contents IDs can be assigned.

7.10

If the s flag is set, then the contents are only meaningful on the same machine/network on which they were saved. This covers filenames and similar things. How to handle checking if this is indeed the same machine is an open question, and beyond the scope of this document. It is certainly true, however, that if the operating system ID does not match the current system and this bit is set, then the chunk should not be copied.

7.11

If the c flag is set, the contents should not be copied when loading and saving a game--they are only relevant to the exact current state of play as stored in the file. The data need not be copied even if this flag is clear, but must not be copied if it is set.

7.12

If the interpreter ID is **____** (four spaces), then the chunk contains information useful to *all* interpreters running on a particular system. This can store a magical OS-dependent reference to the original story file, which need not worry about vagaries of filename handling on more than one system. This chunk may contain anything that can be put in a file and retrieved intact. If the file is restored on a suitable system this can be used to do Good Things.

7.13

If the operating-system ID is **____**, then the chunk contains data useful to *all* ports of a particular interpreter. This may or may not be useful.

7.14

The interpreter and operating-system IDs may not both be **____**. This should not be neccessary.

7.15

If neither ID is **____**, the contents are meaningful only to a particular port of a particular interpreter. Save-file specific preferences probably fall into this category.

7.16

The contents ID will be defined when chunk IDs are picked. Its purpose is to allow multiple chunks to be written containing different data, which is necessary if they need different settings of the c and s flags.

7.17

These extensions add no overhead to interpreters which choose not to handle them, except for larger save files and more chunks to skip when reading files written on another program. Interpreters are not expected to preserve these optional chunks when files are re-saved, although some may be copied, at the option of the interpreter writer or user.

7.18

The only required chunks are **IFhd**, either **CMem** or **UMem**, and **Stks**. The total overhead to a save file is 12 bytes plus 8 for each chunk; in the minimal case (**IFhd**, **[CU]Mem**, **Stks** = 3 chunks), this comes to 36 bytes.

7.19

The following operating system IDs have been registered:

7.19.1

```
'DOS '          MS-DOS (also PC-DOS, DR-DOS)
```

7.19.2

```
'MACS'          Macintosh
```

7.19.3

```
'UNIX'          Generic UNIX
```

7.20

The following interpreter IDs have been registered:

7.20.1

```
'JZIP'          JZIP, the enhanced ZIP by John Holder
```

7.21

The following extension chunks have been registered to date:

System ID	Interp ID	Content ID	Section
'MACS'	' '	0	7.22

7.22

The following chunk has been registered for MacOS, to enable a Macintosh interpreter to find a story file given a save file using the System 7 ResolveAlias call. The MacOS alias record can be of variable size: the actual size can be calculated from the chunk size. Aliases are valid only on the same network as they were saved.

7.22.1

```
4 bytes         'IntD'          chunk ID
```

7.22.2

```
4 bytes         n               chunk length (variable)
```

7.22.3

```
4 bytes         'MACS'          operating system ID: MacOS
```

7.22.4

```
1 byte          00000010        flags (s set; c clear)
```

7.22.5

```
1 byte          0               contents ID
```

7.22.6

```
2 bytes         0               reserved
```

7.22.7

```
4 bytes         '    '          interpreter ID: any
```

7.22.8

```
n-12 bytes      ...             MacOS alias record referencing
                                the story file; from NewAlias
```

7.19.9

Alias records are of variable length, reflected in the chunk length; they are only valid on the same network they were created.

8. Introduction to the IFF format

8.1

This is based on the official IFF standards document, which is rather long and contains much that is irrelevant to the task in hand. I also do not have an electronic copy, so I am including only that which is relevant. Feel free to mail me (i.e. Martin Frost) if there are errors, inconsistencies, or omissions. For the inquisitive, a document containing much of the original standard, including the philosophy behind the structure, can be found at

http://www.cica.indiana.edu/graphics/image_specs/ilbm.format.txt

8.2

IFF stands for "Interchange File Format", and was developed by a committee consisting of people from Commodore-Amiga, Electronic Arts and Apple. It draws strongly on the Macintosh's concept of resources.

8.3

The most fundamental concept in an IFF file is that of a chunk.

8.3.1

A chunk starts with an ID and a length.

8.3.2

The ID is the concatenation of four ASCII characters in the range 0x20 to 0x7E.

8.3.3

If spaces are present, they must be the last characters (there must be no printing characters after a space).

8.3.4

IDs are compared using a simple 32-bit equality test - note that this implies case sensitivity.

8.3.5

The length is a 32-bit unsigned integer, stored in big-endian format (most significant byte, then second most, and so on).

8.4

After the ID and length, there follow (length) bytes of data.

8.4.1

If length is odd, these are followed by a single zero byte. This byte is *not* included in the chunk length, but it is very important, as otherwise many 68000-based readers will crash.

8.5

A simple IFF file (such as the ones we will be considering) consists of a *single* chunk of type **FORM**.

8.5.1

The contents of a **FORM** chunk start with another 4-character ID.

8.5.2

This ID is also the concatenation of four characters, but these characters may only be uppercase letters and trailing spaces. This is to allow the **FORM** sub-ID to be used as a filename extension.

8.6

After the sub-ID comes a concatenation of chunks. The interpretation of these chunks depends on the **FORM** sub-ID (in this proposal, the sub-ID is **IFZS**), except that a few chunk types always have the same meaning (notably the **AUTH**, **(c)_** and **ANNO** chunks described in section 7). For reference, the other reserved types are: **FOR[M1-9]**, **CAT[1-9]**, **LIS[T1-9]**, **TEXT**, and **____** (that is, four spaces).

8.7

Each of these chunks may contain as much data as required, in whatever format is required.

8.8

Multiple chunks with the same ID may appear; the interpretation of such chunks depends on the chunk. For example, multiple **ANNO** chunks are acceptable, and simply refer to multiple annotations. If more than one chunk of a certain type is found, when the reader was only expecting one, (for example, two **IFhd** chunks), the later chunks should simply be ignored (hopefully with a warning to the user).

8.9

Indeed, skipping is the expected procedure for dealing with any unknown or unexpected chunk.

8.10

Certain chunks may be compulsory if the **FORM** is meaningless without them. In this case the **IFhd**, **[CU]Mem** and **Stks** are compulsory.

9. Resources available

9.1

A set of patches exists for the Zip interpreter, adding Quetzal support. They can be obtained from:

http://www.geocities.com/SiliconValley/Vista/6631/

9.2

A utility, **ckifzs** is available as C source code to check the validity of generated save files. A small set of correct Quetzal files are also available. These may be of use in debugging an interpreter supporting Quetzal. These may be obtained from the web page mentioned in 9.1.

9.3

This document is updated whenever errors are noticed or new extension chunks are registered. The latest version will always be available from the above web page. The latest revision designated stable (currently version 1.3) will be in the IF archive, ftp.gmd.de/if-archive, in the directory infocom/interpreters/specification/.

9.4

This document is itself available in a number of forms. The base version is in preformatted ASCII text, but there is also a PDF version (converted by John Holder) and this HTML version (converted by Graham Nelson). Links to all of these may be found on the web page.

9.5

A few interpreters support Quetzal; details will appear here as they become available.

10. Credits

10.1

This standard was created by Martin Frost (email: mdf@doc.ic.ac.uk). Comments and suggestions are always welcome (and any errors in this document are entirely my own, or those of the HTML typesetter, Graham Nelson).

10.2

The following people have contributed with ideas and criticism (in alphabetical order): King Dale, Marnix Klooster, Graham Nelson, Andrew Plotkin, Matthew T. Russotto, Bryan Scattergood, Miron Schmidt, Colin Turnbull, John Wood.

Remarks

Queztal is not a compulsory part of the Z-Machine Standard, since it does not have implications for the behaviour of story files, but it is attached to this copy of the Standard as a highly recommended "optional extra".

Blorb: An IF Resource Collection Format Standard

Version 1.1

Andrew Plotkin <erkyrath@eblong.com>

This is a formal specification for a common format for storing resources associated with an interactive fiction game file. Resources are data which the game can invoke, such as sounds and pictures. In addition, the executable game file may itself be a resource in a resource file. This is a convenient way to package a game and all its resources together in one file.

Blorb was originally designed solely for the Z-machine, which is capable of playing sounds (Z-machine versions 3 and up) and showing images (the V6 Z-machine). However, it has been extended for use with other IF systems. The Glk portable I/O library uses Blorb as a resource format, and therefore so does the Glulx virtual machine. (See http://www.eblong.com/zarf/glk/ and http://www.eblong.com/zarf/glulx/.)

[Text in this document is liberally stolen from Martin Frost <mdf@doc.ic.ac.uk>'s proposal for a common save file format. Ideas are liberally stolen from my own PICKLE format proposal, which is now withdrawn in favor of this proposal.]

This format is named "Blorb" because it wraps your possessions up in a box, and because the common save file format was at one point named "Gnusto". That has been changed to "Quetzal", but I'm not going to let that stop me.

This proposal is longer than I would have liked. However, a large percentage of it is optional stuff -- optional for either the interpreter writer, the game author, or both. That may make you feel better. I've also put in lots of examples, explication, and self-justification.

0: Overall Structure

The overall format will be a new IFF type. The FORM type is 'IFRS'.

The first chunk in the FORM must be a resource index (chunk type 'RIdx'.) This lists all the resources stored in the IFRS FORM.

The resources are stored in the FORM as chunks; each resource is one chunk. They do not need to be in any particular order, since the resource index contains all the information necessary to find a particular resource.

There are five optional chunks which may appear in the file: the color palette (chunk type 'Plte'), the resolution chunk (chunk type 'Reso'), the loop chunk (chunk type 'Loop'), the release number (chunk type 'RelN'), and the game identifier (chunk type 'IFhd'). They may occur anywhere in the file after the resource index.

Several optional chunks may also appear by convention in any IFF FORM: '(c) ', 'AUTH', and 'ANNO'. These may also appear anywhere in the file after the resource index.

1: Contents of the Resource Index Chunk

```
4 bytes         'RIdx'          chunk ID
4 bytes         n               chunk length (4 + num*12)
4 bytes         num             number of resources
num*12 bytes    ...             index entries
```

There is one index entry for each resource. (But not for the optional chunks.) Each index entry is 12 bytes long:

```
4 bytes         usage           resource usage
4 bytes         number          number of resource
4 bytes         start           starting position of resource
```

The index entries should be in the same order as the resource chunks in the file.

The usage field tells what kind of resource is being described. There are currently three values defined:

- 'Pict': Picture resource
- 'Snd ': Sound resource
- 'Exec': Code resource

The number field tells which resource is being described, from the game's point of view. For example, when a Z-code game calls @draw_picture with an argument of 3, the interpreter would find the index entry whose usage is 'Pict' and whose number is 3. For code chunks (usage 'Exec'), the number should contain 0.

The start field tells where the resource chunk begins. This value is an offset, in bytes, from the start of the IFRS FORM (that is, from the start of the resource file.)

2: Picture Resource Chunks

Each picture is stored as one chunk, whose content is either a PNG file or a JPEG (JFIF) file. The chunk type is 'PNG ' or 'JPEG'. (Note that these are two possible formats for a single resource. It is not possible to have a PNG image and a JPEG image with the same image resource number.)

The PNG file format is available at

http://www.cdrom.com/pub/png/

For information on JPEG, see

http://www.jpeg.org/public/jpeghomepage.htm

3: Sound Resource Chunks

Each sound is stored as one chunk, whose content is either an AIFF file, a MOD file, or a song file. (Note that these are three possible formats for a single resource. It is not possible to have an AIFF sound and a MOD sound with the same sound resource number.)

An AIFF (Audio IFF) file has chunk type 'FORM', and formtype 'AIFF'. The AIFF format is available at these locations:

http://developer.apple.com/techpubs/mac/Sound/Sound-61.html

ftp://ftp.sgi.com/sgi/aiff-c.9.26.91.ps.Z

A MOD file has chunk type 'MOD '. This is the Amiga-originated format for music synthesized from note samples. The specification, such as it is, is available in the files

http://www.eblong.com/zarf/blorb/mod-spec.txt

For generality, MOD files in Blorb are limited to the ProTracker 2.0 format: 31 note samples, up to 128 note patterns. The magic number at byte 1080 should be 'M.K.' or 'M!K!'.

A song file has chunk type 'SONG'. This is similar to a MOD file, but with no built-in sample data. The samples are instead taken from AIFF sounds in the resource file. For each sample, the 22-byte sample-name field in the song should contain the string "SND1" to refer to sound resource 1, "SND953" to refer to sound resource 953, and so on. Any sound so referred to must be an AIFF, not a MOD or song. (You can experiment with fractal recursive music on your own time.)

Each sample record in a MOD or song contains six fields: sample name, sample length, finetune value, volume, repeat start, repeat length. In a MOD file, the sample name is ignored by Blorb (it is traditionally used to store a banner or comments from the author.) In a song file, the sample name contains a resource reference as described above; but the sample length, repeat start, and repeat length fields are ignored. These values are inferred from the AIFF resource. (The repeat start and repeat length are taken from the sustainLoop of the AIFF's instrument chunk. If there is no instrument chunk, or if sustainLoop.playMode is NoLooping, there is no repeat; the repeat start and length values are then considered zero.)

Note that an AIFF need not contain 8-bit sound samples, as a sound built into a MOD would. A clever sound engine may take advantage of this to generate higher-quality music. An unclever one can trim (or pad) the AIFF's data to 8 bits before playing the song. In the worst case, it is always possible to trim the AIFF data to 8 bits, append it to the song data, fill in the song's sample records (with the appropriate lengths, etc, from the AIFF data); the result is a complete MOD file, which can then be played by a standard MOD engine.

The intent of allowing song files is both to allow higher quality, and to save space. Note samples are the largest part of a MOD file, and if the samples are stored in resources, they can be shared between songs. (Typically note samples will be given high resource numbers, so that they do not conflict with sounds used directly by the game. However, it is legal for the game to use a note sample as a sampled-sound effect, if it wants.)

On the Z-machine, there are also the problems of how the game knows the interpreter can play

music, and how sampled sounds are played over music. See the section "Z-Machine Compatibility Issues" later in this document. (These issues are not relevant to Glk and Glulx.)

4: Executable Resource Chunks

There should at most one chunk with usage 'Exec'. If present, its number must be zero; its content is a Z-code or Glulx game file, and its chunk type is 'ZCOD' or 'GLUL'.

[Other executable formats may be added in the future. Some of them might support multiple executable chunks; for example, a VM which supports concurrent processes. In such a case, chunk zero should contain the code to execute first, or at the top level.]

A resource file which contains an executable chunk contains everything needed to run the executable. An interpreter can begin interpreting when handed such a resource file; it sees that there is an executable chunk, loads it, and runs it.

A resource file which does not contain an executable chunk can only be used in tandem with an executable file. The interpreter must be handed both the resource file and the executable file in order to begin interpreting.

If an interpreter is handed inconsistent arguments -- that is, a resource file with no executable chunk, or a resource file with an executable chunk plus an executable file -- it should complain righteously to the user.

5: The Release Number Chunk

This chunk is used to tell the interpreter the release number of the resource file. It is meaningful only in Z-code resource files.

The interpreter passes this information to the game when the @picture_data opcode is executed with an argument of 0. The release number is a 16-bit value. The chunk format is:

```
4 bytes         'RelN'          chunk ID
4 bytes         2               chunk length
2 bytes         num             release number
```

This chunk is optional. If it is not present, the interpreter should assume a release number of 0.

6: The Game Identifier Chunk

This identifies which game the resources are associated with. The chunk type is 'IFhd'.

This chunk is optional. If it is present, and the interpreter is given a game file along with a resource file, the interpreter can check that the game matches the IFhd chunk. If they don't, the interpreter should display an error. The interpreter may want to provide a way for the user to ignore or skip this error (for example, if the user is a game author testing changes to the game file.)

If the resource file contains an executable chunk, there's not much point in putting in the IFhd chunk.

For Z-code, the contents of the game identifier chunk are defined in the common save file format specification, section 5. This spec can be found at

`http://www.ifarchive.org/if-archive/infocom/interpreters/specification/savefile_14.txt`

The "Initial PC" field of the IFhd chunk (bytes 10 through 12) has no meaning for resource files. It should be set to zero.

For Glulx, the contents of the game identifier chunk are defined in the Glulx specification. This can be found at http://www.eblong.com/zarf/glulx/.

7: The Color Palette Chunk

This contains information about which colors are used by picture resources in the file. The chunk type is 'Plte'. It is optional, and should not appear if there are no 'Pict' resources in the file.

The format is:

```
4 bytes         'Plte'          chunk ID
4 bytes         n               chunk length
n bytes         ...             color data
```

There are two possibilities for the color data format. The first is an explicit list of colors. In this case, the data consists of 1 to 256 color entries. Each entry is three bytes, of the form:

```
1 byte          red value (0 = black, 255 = red)
1 byte          green value (0 = black, 255 = green)
1 byte          blue value (0 = black, 255 = blue)
```

The second case is a single byte, which may have either the value 16 or 32 (decimal). 16 indicates that the picture resources are best displayed on a direct-color display which has 16 or more bits per pixel (5 or more bits per color component.) 32 indicates that the resources are best displayed with 32 or more bits per pixel (8 or more bits per color component.)

The two cases are differentiated by checking the chunk length (n). If n is 1, it's a direct color value; if it's a positive multiple of 3, it's a color list, and the number of entries is the length divided by 3. Any other length is illegal.

This chunk is only a hint; there is no guarantee about what the interpreter will do with it. A color list will most likely be useful if the interpreter's display can only display a limited number of colors (for example, an 8-bit indexed color device). The interpreter may set the display to the colors listed in the palette. Or it may set the display to just some of the colors listed (for example, if it wishes to reserve some colors for text display, or if it just doesn't have enough colors available.) Or the interpreter may ignore the palette chunk, or do something else.

Similarly, if the interpreter finds a "16" or "32" value, it may set the display to the appropriate bit depth. Or it may set the display to an 8-bit color cube, and dither the images for display. Or, again, it may ignore the palette chunk entirely, or do something else.

It is not required that the palette chunk list every color used in the 'Pict' resources. It is not required that the colors in the palette all be different, or that they all are actually used by 'Pict' resources. It is not required that the palette have anything to do with the game art at all. Of course, if you give the interpreter misleading hints, you deserve whatever you get.

8: The Resolution Chunk

This chunk contains information used to scale images. The chunk type is 'Reso'. It is optional. This chunk is meaningful only in Z-code resource files.

A scalable image is one which the author says should be larger when more space is available, and smaller when less space is available. (Note that the Z-code game does *not* directly control the scaling of images. The interpreter controls the scaling of images, in response to the information in the resolution chunk. The interpreter then provides the scaled size in response to @picture_data queries, and the game draws its display based on those queries.)

It is also possible to create images that have a fixed scaling ratio; they are always scaled up or down by a particular amount, regardless of window size.

Not all images have to be scalable. Unless the resolution chunk gives scaling data for an image, that image is assumed to be non-scalable. Non-scalable images are always displayed at their actual size. (One image pixel per screen pixel.)

This chunk is optional; if it is not present, then all of the images in this file are non-scalable.

```
4 bytes         'Reso'          chunk ID
4 bytes         num*28+24       chunk length
4 bytes         px              standard window width
4 bytes         py              standard window height
4 bytes         minx            minimum window width
4 bytes         miny            minimum window height
4 bytes         maxx            maximum window width
4 bytes         maxy            maximum window height
num*28 bytes    ...             image resolution entries
```

The "standard window size" is the normal size, the author's original chosen size, for the Z-machine window. It is not the only possible size; a good V6 game should be prepared for any window the interpreter chooses to create. The idea is that when the Z-machine window is exactly the standard size, scalable images are presented at their original size. When the Z-machine window is larger than the standard size, scalable images are scaled up; when it is smaller, scalable images are scaled down.

The minimum and maximum window sizes are provided as a hint to the interpreter, when it is choosing a window size. It may also use the standard window size as a hint for this purpose. (If the interpreter lacks the ability to choose its own window size, of course, it will ignore these hints.) The idea is that the minimum and maximum sizes define the range in which the game can draw itself successfully.

Any or all of minx, miny, maxx, maxy can indicate "no limit in this direction" by containing a value of zero. However, px and py must contain non-zero values. Unless the min or max values are zero, it must be true that minx <= px <= maxx, miny <= py <= maxy.

Important note: The standard, minimum, and maximum window size values are measured in *screen pixels*. Furthermore, unscaled pictures should be drawn in screen pixels -- one image pixel per screen pixel. (This may seem dumb as rocks, and maybe it is, but my rationale is presented at the end of this document.)

Also note that I have not mentioned Z-pixels. This standard does not concern itself with Z-pixels.

On with the show.

The standard, minimum, and maximum window sizes are followed by a set of image entries, one for each scalable image. (Non-scalable images do not have an entry in this table; that's how they are declared to be non-scalable.) Each image entry is 28 bytes, of the form:

```
4 bytes         number          image resource number
4 bytes         ratnum          numerator of standard ratio
4 bytes         ratden          denominator of standard ratio
4 bytes         minnum          numerator of minimum ratio
4 bytes         minden          denominator of minimum ratio
4 bytes         maxnum          numerator of maximum ratio
4 bytes         maxden          denominator of maximum ratio
```

The number is the picture number; in other words, this entry applies to the resource whose usage is 'Pict' and whose number matches this value.

The entry then contains a standard, minimum, and maximum image scaling ratio. Each ratio is a real number, represented by two integers:

- stdratio = ratnum / ratden
- minratio = minnum / minden,
- maxratio = maxnum / maxden.

Minratio can indicate zero ("no minimum limit") by having both minnum and minden equal to zero. Similarly, maxratio can indicate infinity ("no maximum limit") by having maxnum and maxden equal to zero. It is illegal to have only half of a fraction be zero.

To compute the actual scaling ratio for this image, the interpreter must first compute the overall game scaling ratio, or Elbow Room Factor (ERF). If the actual game window size is (wx,wy), and the standard window size is (px,py), then

- ERF = (wx/px) or (wy/py), whichever is smaller.

(Note that if the game's window is exactly its standard size, ERF = 1.0. If the window is twice the standard size, ERF = 2.0. If the window is three times the standard width and four times the standard height, then ERF = 3.0, because there's really only enough room for the game's standard layout to be tripled before it overflows horizontally.)

The scaling ratio R for this image is then determined:

- If ERF*stdratio < minratio, then R = minratio.
- If ERF*sdtratio > maxratio, then R = maxratio.
- If minratio <= ERF*stdratio <= maxratio, then R = ERF*stdratio.

If minratio and maxratio are the same value, then R will always be this value; ERF and stdratio are ignored in this case. (This indicates a scalable image with a fixed scaling ratio.)

The interpreter then knows that this image should be drawn at a scale of R screen pixels per image pixel, both vertically and horizontally. The interpreter should report this scaled size to the game if queried with @picture_data (as opposed to the original image size).

Yes, this is an ornate system. The author is free to ignore it by not including a resolution chunk. If the author wants scaled images, or variably-scalable images, this system should suffice.

Here are some examples. They're not necessarily examples of good art design, but they do demonstrate how a given set of desires translate into images and resolution values. All are for a game with a standard size of (600,400).

The game wishes a title image that covers the entire window, and all the resolution should be visible at the standard size. (So if the window is twice the standard size, the image will be stretched and coarse-looking; if the window is half the standard size, the image will be squashed and lose detail.)

- Image size (600,400); stdratio 1.0; minratio zero; maxratio infinity.

The game has a background image of a cave, made from a scanned photograph. At standard window size, this should cover the entire window, but not all the detail needs to be visible. If the window is larger, the image should still cover the entire window; more detail will be visible, up to twice the standard size (at which point all the resolution should be visible.) If the window is larger than twice the standard size, the image should not be stretched farther; instead the game will center it and have blank space around the edges.

- Image size (1200,800); stdratio 0.5; minratio zero; maxratio 1.0.

The game has small monochrome icons indicating different magical perceptions, which it will draw interspersed with the text. The icons should always be drawn at double size, two screen pixels per image pixel, regardless of the window size.

- Image size (20,20); stdratio 1.0; minratio 2.0; maxratio 2.0. (In this case, remember, the stdratio value is ignored.)

The game has a graphical compass rose which it will draw in the top left corner. This should be 1/4 of the window size in the standard case, and shrink proportionally if the window is smaller. However, if the window is larger than standard, the rose should not grow; all the extra space can be allotted for text. All detail (image pixels) should be visible in the standard case.

- Image size (150,100); stdratio 1.0; minratio zero; maxratio 1.0.

The same compass rose, still to be 1/4 of the window size -- but this time it is critical that all the image detail be visible when the window is as small as half-standard (that is, when the rose is 75 by 50 pixels). At standard scale (150 by 100), it will therefore appear stretched and coarse. If the window is smaller than half the standard size, the rose should not shrink beyond 75x50, so that pixels are never lost.

- Image size (75,50); stdratio 2.0; minratio 1.0; maxratio 2.0.

End of verbose examples.

9: The Looping Chunk

This chunk contains information about which sounds are looped, in a V3 Z-machine game. The chunk type is 'Loop'. It is optional.

Note that in V5 and later, the @sound_effect opcode determines whether a sound loops. The looping chunk is ignored. Therefore, this chunk should not be used at all in Blorb files intended for games which are not V3 Z-machine games.

The format is:

```
4 bytes         'Loop'          chunk ID
4 bytes         num*8           chunk length
num*8 bytes     ...             sound looping entries
```

Each entry is 8 bytes, of the form:

```
4 bytes         number          sound resource number
4 bytes         value           repeats
```

The repeats flag is one if the sound is to be played once; it is zero if the sound is to repeat indefinitely (until it is stopped or another sound started.) If there is no entry for a particular sound resource, or if the looping chunk is absent, the V3 interpreter should assume the flag is one, and play the sound exactly once.

10: Other Optional Chunks

A resource file can contain extra user-level information in 'AUTH', '(c) ', and 'ANNO' chunks. These are all optional. An interpreter should not do anything with these other than ignore them or (optionally) display them.

These chunks all contain simple ASCII text (all characters in the range 0x20 to 0x7E). The only indication of the length of this text is the chunk length (there is no zero byte termination as in C, for example).

The 'AUTH' chunk, if present, contains the name of the author or creator of the file. This could be a login name on multi-user systems, for example. There should only be one such chunk per file.

The '(c) ' chunk contains the copyright message (date and holder, without the actual copyright symbol). There should only be one such chunk per file.

The 'ANNO' chunk contains any textual annotation that the user or writing program sees fit to include.

11: Z-Machine Compatibility Issues

The image system presented in this document is fully backwards-compatible with Infocom's interpreters. Infocom V6 games, such as Arthur, Journey, and Zork Zero, contain only non-scalable image resources. The game files are written to deal with both variations in window size and variations in image size (since the interpreters for different platforms had different window sizes and different art.) Therefore, if you construct a Blorb file containing the images from a particular platform (say, the Mac) and give it the suggested window size of the Infocom Mac interpreter, the game file will deal with it correctly.

The image system is also sort of forwards-compatible, in the following sense. If you take a Blorb file whose standard (intended) window size is the same as the Infocom interpreter's, and break it out into Infocom image files, the Infocom interpreter should display it correctly. The interpreter will not scale images, but since the window size is equal to the standard size, the Blorb rules require the images to be displayed unscaled anyway.

Also, of course, if you take a Blorb file which contains only non-scalable images, an Infocom interpreter will act correctly, since it will not scale the images regardless of the standard size.

The sound system is slightly more problematic. A game file can announce that it uses sound, by setting a header bit; the interpreter can announce that it does not support sound, by clearing that bit. But there is no way to distinguish a game that uses sampled sound only, from one that uses sampled sound and music. (And similarly for the interpreter's support of samples versus music.) This may be addressed in a future revision of the Z-machine. In the meantime, games should set that header bit if any kind of sound is used (samples or music or both.) And interpreters should clear that bit only if *no* sound support is available. If the interpreter supports sampled sound but not music, it should leave the header bit set, announcing that it does "support sound." It should then ignore any request to play a music resource.

There is also the question of overlapping sounds. The Z-Spec (9.4.2) says that starting a new sound effect automatically stops any current one. But it is not desirable that a sound effect such as footsteps should interrupt the playing of background music. Therefore, the interpreter should amend this rule, and consider sampled sounds and music to be in seperate "channels". Samples interrupt samples, and music interrupts music, but one form of sound does not interrupt the other.

This is an actual variance in the behavior of the Z-machine, and worse, a variance which depends on data format. (One sound will either stop another, or not, depending on whether the sound is stored in AIFF or MOD format.) We apologize for the ugliness.

Again, future versions of the Z-machine may address this issue, and allow a more general system where any sound can be overlaid on any other sound, or interrupt it, as the game desires and regardless of storage format. (After all, there can be background *sounds* as well as background *music*.) Such a system would also allow the interpreter to announce its limitations and capabilities -- whether it can play music, whether it can play two pieces of music at once, how many sampled sounds it can play at once, etc.

A final, ah, note: The remark at the end of Z-Spec chapter 9, about sequencing sound effects to simulate the slow Amiga version of "The Lurking Horror", should not be applied to music sounds. New music should interrupt old music immediately, regardless of whether keyboard input has occurred since the old music started.

12: Glk Compatibility Issues

The Glk I/O library was designed with portable resources in mind, so there should be no incompatibility.

Remember that the resolution and scaling data is not used by Glk. That chunk is ignored by Blorb-capable Glk libraries.

13: The IFF Format

A description of the IFF format can be found at

http://www.cica.indiana.edu/graphics/image_specs/ilbm.format.txt

In the interests of simplicity, this proposal does not use IFF LISTs or CATs, even though its purpose is to contain concatenated lists of data. Therefore, the format can be quickly described as follows:

```
4 bytes         'FORM'          Magic number indicating IFF
4 bytes         n               FORM length (file length - 8)
4 bytes         'IFRS'          FORM type
n-4 bytes       ...             The chunks, concatenated
```

Each chunk has the following format:

```
4 bytes         id              Chunk type
4 bytes         m               Chunk length
m bytes         ...             Chunk data
```

If a chunk has an odd length, it *must* be followed by a single padding byte whose value is zero. (This padding byte is not included in the chunk length m.) This allows all chunks to be aligned on even byte boundaries.

All numbers are two-byte or four-byte unsigned integers, stored big-endian (most significant byte first.) Character constants such as 'FORM' are stored as four ASCII bytes, in order from left to right.

When reading an IFF file, a program should always ignore any chunk it doesn't understand.

14: Other Resource Arrangements

It may be convenient for an interpreter to be able to access resources in formats other than a resource file. In particular, when developing a game, an author will want to load images and sounds from individual files, rather than having to re-package all the resources whenever any one of them changes.

Such resource arrangements are platform-specific, and the details are left to the interpreter. However, one suggestion is to have a single directory which contains all the resources as files, with one file per resource. (PNG files for images, and so on. The contents of each file would be exactly the same as the contents of the equivalent chunk, minus the initial eight bytes of type/length information.) Files would be named something like "PIC1", "PIC2"..., "SND1", "SND2"..., and so on. A Z-code file (if present) would be named "STORY"; a color palette would be named "PALETTE", a resolution chunk "RESOL", a looping chunk "LOOPING", a release number "RELEASE", and a game identifier chunk "IDENT". (Naturally, file suffixes would be added in platforms that require them.) The interpreter would be started up and handed the entire directory as an argument; or possibly the directory along with a separate Z-code file.

15: Rationales and Rationalizations

- *Why have a common resource collection format?*

Infocom chose not to standardize their resource formats; they had a different picture format for each platform. This was a reasonable choice for them, since they were writing all the games, all the art, and all the interpreters. They therefore had the capacity to translate the art into platform-specific formats for all the platforms they supported.

In the modern age, an IF author does not have access to all the platforms his game will be played on. It is therefore reasonable to distribute art in a single format, and leave interpreter writers the job of supporting that format.

- *Why an IFF-based format?*

IFF does what we want; it's a known, very simple way to concatenate chunks of data together.

Also, the common save-file format is IFF-based. This allows interpreters to use the same code for reading both save files and resource files.

- *Why not compress data as well as archiving it? Why just concatenate everything together as chunks?*

Any reasonable sound or image format already incorporates compression.

- *Why is there a "number" field in the entries in the resource index chunk? Why not just assume chunks are numbered consecutively?*

On the Z-machine, pictures are not necessarily numbered contiguously (Z-Spec 8.8.6.1.) Sounds are numbered consecutively, but sounds 1 and 2 are bleeps, so the game-specific sound resources start at 3. (Z-Spec 9.2.) In Glk, resources need not be contiguous at all. Rather than jigger the numbering or require place-holder chunks, I decided there should be an index which maps resource numbers to chunks.

- *Why only two image format? Why not allow any image format?*

The whole point of this exercise is to assure the author that the player can view his art. If we allow lots of different formats, we can't possibly insist that every interpreter must display all the formats. This leaves us just about back where we started. Individual game authors would be negotiating with individual interpreter authors to support particular formats, and it would just be icky.

Therefore, we *do* insist that every interpreter be able to display all the formats listed in this standard. That means a small number of formats. See the next two questions.

- *Okay, then, why three sound formats?*

Because a sampled-sound format (like AIFF) can reproduce anything; and a note format (like MOD) can reproduce music with much less data than AIFF. It's effectively a very efficient form of compression which only works for musical sound. Song files are even more efficient, if you have several songs, because the note samples are shared. But standard MOD playing and composition tools work on MODs, so it wouldn't make sense not to allow those too. Sigh.

- *Why PNG and JPEG for images?*

The PNG format is not burdened with patent restrictions; it is free; it's not lossy; and it can efficiently store many types of images, from 1-bit (monochrome) images up to 48-bit color images. JPEG is lossy and not optimal for images other than photographs, but compresses photographs

well. Earlier versions of Blorb specified only PNG, but JPEG was a popular request, and the two formats should complement each other.

As to other possibilities: GIF is a popular format, but it is owned by twits who restrict its use. TIFF has been suggested, but it seems to be overly baroque. Blorb is likely to stay with PNG and JPEG for the foreseeable future.

- *What is the Blorb Policy on Color Depth?*

The idea is that each author can decide what kind of color requirements his game will have.

The alternative (which we did *not* choose) would be to mandate a fixed set of color requirements for all graphical games -- for example, an 8-bit color display set to a color-cube set of colors. This seems like a dumb idea. Any fixed set of requirements is going to be impossible for some machines and standard equipment on others, and both these sets will change over time. The requirements would quickly become obsolete.

Instead, we choose to allow any kind of art in graphical games. If the author includes only monochrome images, the game will run anywhere. If the author includes full-color 32-bit images, he is creating a game which wants a powerful graphics machine to display itself on. That's the author's choice. If the player's machine only allows 8-bit color, his interpreter will have to dither or otherwise reduce the color information of the game art. The player can accept this, buy a more powerful computer, or throw away the game. There's no way around that. The problem can only be avoided by outlawing 32-bit color images, which we do not wish to do.

- *What's the idea of the palette chunk?*

The palette chunk itself is provided for the benefit of interpreters which can control their display palettes or color depth. The palette declares the minimum set of colors (or direct-color depth) which the author wants you to have in order for the game to "look okay." It may be a good idea to switch palettes in order to play a particular game; the palette chunk tells the interpreter this advice.

Now, the interpreter is not *required* to follow this advice. This is for the player's benefit; if the player has a monochrome machine, or just doesn't like changing palettes, he is not denied the opportunity to play the game. He'll just get reduced-quality art. That's his choice. As stated above, he can accept it, upgrade, or throw the game away.

- *What is the Blorb Policy on Pixel Size?*

We make a couple of assumptions.

Assumption one: Image pixels are square. Your images should have the correct aspect ratio when drawn with square pixels -- that is, when the number of image-pixels-per-inch is the same vertically and horizontally. If your art program doesn't understand square pixels, get a real art program. There. That's resolved.

(This means that if an image is 400 pixels wide and 200 pixels high, the interpreter should draw it on the screen with a physical width twice its physical height. Anything else will look distorted.)

(It has been noted that this does not exactly apply to Infocom's V6 games; their art was probably designed for an era of computers that did not have exactly square pixels (IBM EGA, Apple II machines displaying on television screens, and other such barbarisms.) However, this does not seem to have concerned them. Infocom interpreters which are running on modern machines, with square-pixel displays, display their art with square pixels. We will do the same.)

Assumption two: It is always okay to draw images at their "actual size" -- one image pixel per screen pixel.

Now you think I'm crazy. It is true that many modern screens can be adjusted to different pixel sizes; mine allows 55, 72, or 88 pixels per inch. However, *I declare this to be an illusion.* If a user sets his monitor to smaller pixels, it's because he wants a given image to be smaller. So he can fit more of them on screen. He also wants his text to be smaller, and his windows. That's the way web browsers work, that's the way Adobe Photoshop works, and that's damn well good enough for the Z-machine.

Perhaps in the future there will be monitors that break this rule -- much smaller pixels, 300 or 600 pixels per inch, for example. At that time there will be some consensus on how to display images. (Frankly, I expect it will be "draw them at 55, 72, or 88 pixels per inch, depending on the user's previous preference.") Z-machine interpreters can follow that plan when it emerges.

Until then, the right size for a non-scaled picture is one image pixel per screen pixel. If an image is to be scaled by a ratio of 2.0, then the right size for it is one image pixel per two screen pixels (vertically and horizontally). And so on.

- *Where do Z-pixels come into all this?*

The definition of Z-pixels is entirely up to the interpreter. This standard says nothing on the subject, and does not care.

It is true that the interpreter must tell the game what the window size and image sizes are, as measured in Z-pixels. That's the interpreter's job. The interpreter knows how big its window is, in screen pixels; it translates that into Z-pixels -- using whatever definition it has -- and reports it to the game. Then, the scaling rules of this spec define what the display sizes of the images will be, as measured in screen pixels. The interpreter translates these sizes into Z-pixels -- using the same definition -- and reports them to the game. All consistent and well-defined.

- *What is the Blorb Policy on Interpreters that do Funky Stuff?*

The interpreter is Allowed. It's okay to be ugly.

For example, someone may (in a fit of insanity) write a Blorb-compliant interpreter for the Apple II. The Apple II had non-square pixels. But (assume) it doesn't have the processing power to scale all its images by a factor of 1.2 (or whatever) to adjust for this. Well, it's legal to write an interpreter that draws art at one image pixel per (non-square) screen pixel. The art will look distorted; the user can like it or play on a different machine.

For another example, someone may want their entire game display doubled in size. All the art twice as large (in screen pixels) as this spec says it should be. An interpreter which has this option is legal. It's the moral equivalent of mounting a magnifying glass on your monitor -- that certainly doesn't violate any software standards.

- *What about playing Blorb-packaged games on original Infocom interpreters?*

It's possible. You'll have to unpack the PNG art and translate it into the format that Infocom used. Since the Infocom interpreters had a hard-wired screen size, you can precompute all the scaling factors, and do any necessary scaling in the translation process. 32-bit color images will have to be color-reduced; that's the way it goes. But the result should be fully playable on Infocom's interpreters.

- *Have you considered --*

Yes.

Z-Machine Standard 1.1 Proposal

(*Draft 6*)

Authors: Kevin Bracey & Jason C. Penney

(*The latest version of this document is linked at* http://www.inform-fiction.org/zmachine/standards/ *or - if that fails - available at* http://www.jczorkmid.net/~jpenney/)

Introduction

This revision of the Z-Machine Standards Document has been written with the following aims:

1. To clarify problem areas in Standard 1.0, and correct errors
2. To allow more probing of interpreter capabilities
3. To clarify the use of Blorb with Standard 1.0
4. To add some extensions for use with Blorb
5. To add some simple extensions to the Version 5 and Version 6 screen models to increase flexibility
6. To allow easier access to arbitrary Unicode characters

As ever, interpreters **fully** implementing this Standard (apart from any optional parts whose absence are signalled through header bits) for a given Version shall indicate this by setting the Standard revision bytes in the header to $01 $01. (An interpreter might meet the Standard for all Versions other than V6, in which case it would not fill in the revision bytes when playing a V6 game).

There have been instances in the past of interpreters that do not fully meet Standard 1.0 claiming Standard 1.0 compliance in the header. This sort of behaviour is dangerous, and likely to cause games that would otherwise work to break.

Conversely, games that absolutely require Standard 1.1 features should not refuse outright to run on a non-Standard 1.1 interpreter - it may be implementing the new features you require, being deficient only in other areas. Instead, a warning should be issued to the user before proceeding. Of course, whereever possible games should have alternative behaviour automatically invoked for older interpreters.

As with the rest of the Standard, all references to Version 5 in this document apply equally to Versions 7 and 8.

The word SHOULD in this document denotes a strong recommendation - implementors may take other action if they have a good reason.

Other specific phrasing, such as MUST, SHALL, WILL, DOES or ARE, indicates an absolute requirement of the Standard. Any deviation indicates non-compliance.

Updates / Clarifications

Memory layout

Version 6 and 7 story files are both limited to 512K, not 320K. The 320K limitation was an artificial one imposed by the Inform compiler - this has now been addressed by a compiler patch.

Character set

In the past, not just in the Z-machine world, there has been general confusion over the rendering of ASCII/ZSCII/Latin-1/Unicode characters $27 and $60. For the Z-machine, the traditional interpretations of right-single-quote/apostrophe and left-single-quote are preferred over the modern neutral-single-quote and grave accent - see Table 2A of the Inform Designer's Manual. $22 is a neutral double-quote.

An alternative rendering is to interpret both $27 and $60 as neutral quotes, but interpreting $60 as a grave accent is to be avoided.

No doubt aware of this confusion, Infocom never used character $60, and used $27 almost exclusively as an apostrophe - hardly any single quotes appear in Infocom games. Modern authors would do well to follow their lead.

The few Infocom games that do use single quotes use $27 for both opening and closing - but even on many of their interpreters this looked a little odd, so suggesting that $27 be a right quote introduces no extra compatibility problems.

Notes on smartening output

Interpreters can easily convert $22 into opening and closing double quotes automatically if they desire. Adjustment of $27 and $60 is ill-advised, due to the difficulty in distinguishing single-quotes from apostrophes – witness the following examples from various Infocom games:

```
"I'm Wiggins, 'ead o' the Baker Street Irregulars."
... a majority of people agree our school system is 'out of control.'"
... agreed that 2051, '61, and '71 all looked disturbing. ...
[Too bad there's no scratch 'n' sniff for this one, huh?]
The word 'examne' isn't in your vocabulary.
```

Thus if authors want to have smart single quotes, it is up to them to correctly use opening ($60) and closing ($27) forms.

One possible heuristic for an interpreter to elegantly handle both types of quoting would be to initially treat $27 as a neutral quote, except when occurring between two letters, when it is clearly an apostrophe. As soon as a $60 is printed, it is clear that the game is intending to use smart quotes,

so $27 from then on should always be a right quote.

There is a similar problem with character $2D, which could be hyphen, minus or a dash. When proportional fonts are used these forms should look very different, so $2D is a candidate for interpreter smartening. By default it should be assumed to be a hyphen, but conventional forms

worth spotting are " - " to represent an en-dash and " -- " to represent an em-dash, although such smartening mustn't be applied when the game has requested fixed-pitch. "-" immediately preceding a digit could be converted to a minus.

If an interpreter is able to output such typographical characters, it should also ensure that they are available manually via the appropriate Unicode code points (eg U+2018 and U+2019 for left and right single quotes), and that `@check_unicode` reports their presence or absence correctly.

Encoded text

In Version 3 and later, many of Infocom's interpreters (and some subsequent interpreters, such as ITF's) treat two consecutive Z-characters 4 or 5 as shift locks, contrary to the Standard. As a result, story files should not use multiple consecutive 4 or 5 codes except for padding at the end of strings and dictionary words. In any case, these shift locks are not used in dictionary words, or any of Infocom's story files.

Two extra rules apply to encoding text for dictionary words in Versions 1 and 2 only (see 3.7). If the truncation to 6 Z-characters leaves a multi-Z-character construction incomplete, the end-bit of the last word is not set. Also, shift-lock Z-characters 4 and 5 are used instead of the single-shift Z-characters 2 and 3 when the next two characters come from the same alphabet. These two rules affect the dictionary of Zork I, which contains "nasty-" (looking), "storm-" (tossed) and "pdp10". Arguably, the first rule could be a bug, but the 3 extant V1 and V2 files are consistent. The games are still winnable without access to those dictionary words.

Unicode

The Z-machine is not expected to handle complex Unicode formatting like combining characters, bidirectional formatting and unusual line-wrapping rules - it remains firmly based in the world of left-to-right text with space breaks between words - every character is viewed as separate spacing glyph.

All output is in presentation order, from left to right. To handle languages like Arabic or Hebrew, text would have to be output "visually", with manual line breaks (possibly via an in-game formatting engine).

Far eastern languages are generally straightforward, except they usually use no spaces, and line wraps can occur almost anywhere. The easiest to way to handle this would be for the game to turn off buffering. A more sophisticated game might include its own formatting engine. Also, fixed-space output is liable to be problematical with most Far Eastern fonts, which use a mixture of "full width" and "half width" forms - all half-width characters would have to be forced to full width.

The only way to output arbitrary Unicode characters in Standard 1.0 is one character at a time with the `@print_unicode` opcode. Standard 1.1 adds Unicode string printing - see below.

The Z-machine does not provide access to non-BMP characters (ie characters outside the range U+0000 to U+FFFF).

Unicode characters U+0000 to U+001F and U+007F to U+009F are control codes, and must not be used.

Header capabilities bits

The (Version 6) sound and graphics bits in Flags 1 indicate general availability of sound and graphics - ie whether the associated opcodes are available and functional.

The bits in Flags 2 and 3 should ideally be set reflecting current availability, rather than general support. In other words, if no Blorb (or other) resources for this story file have been found, or if the Blorb file contains no graphics or no sound, the corresponding bits should be cleared.

Also, it is recommended that interpreters that would prompt for an auxiliary Blorb file should do so immediately on start up if any of the "game wants to use sound/music/graphics" bits are set; this allows the bits to be cleared if no file is forthcoming, before the game starts execution. The game can then take appropriate action.

Bit 4 of Flags 1 indicates the availability of the fixed-pitch style, not Font 4.

The fixed-pitch header bit

Use of the fixed-pitch bit in the header is deprecated in Version 5 and later - it is an irregularity in the Z-machine that complicates the implementation of buffering and text style handling in the interpreter, and may even slow down all memory stores.

Font 4 (or Fixed Pitch style) should be used instead. Infocom did not use the fixed-pitch bit after Version 4, and it is not implemented in at least some of their later interpreters.

However, the fixed-pitch bit is modified by the "font" statement in Inform 6.21, so is used by many current Inform generated files. Therefore interpreter authors must support it in Version 5. Support in Version 6 is not required.

Indirect variable references

In the seven opcodes that take indirect variable references (`inc`, `dec`, `inc_chk`, `dec_chk`, `load`, `store`, `pull`), an indirect reference to the stack pointer does not push or pull the top item of the stack - it is read or written in place.

`@art_shift` and `@log_shift`

The "places" operand must be in the range -15 to +15, otherwise behaviour is undefined. (This is because most interpreters will be implementing the opcodes using C's << and >> operators, and this is the range of ISO C standard-defined behaviour for shifts of a 16-bit type).

Version 6 windows

Interpreter authors are advised that all 8 windows in Version 6 must be treated identically. The only ways in which they are distinguished are:

- Different default positions + sizes
- Different default attributes
- `@split_window` manipulates windows 0 and 1 specifically
- Window 1 is the default mouse window

Differences in interpreter behaviour must only arise from differences in window attributes and properties.

@set_text_style

It was previously unclear in the Standard whether instructions like

```
@set_text_style 5;   ! Reverse+Italic
```

were legal. None of Infocom's game files make use of such forms, and many interpreters will not handle them correctly. Such instructions should be avoided; instead you can use multiple `@set_text_style` opcodes specifying each style in turn, with the most important last, thus:

```
@set_text_style 4;   ! Italic
@set_text_style 1;   ! Reverse - overrides if combination not available
```

However, Standard 1.1 does now require such combinations in a single opcode to be supported, as they are in some of Infocom's later interpreters – see below. But unless a game absolutely requires Standard 1.1 anyway (eg for true colour support), it is probably best to request only one style per opcode for backwards compatibility.

@set_font

Font 2 (the picture font) is undefined. No Standard interpreter will implement it, and `set_font 2` will always return 0. Any new fonts will have numbers higher than 4. Fonts 5-1023 are reserved for future Standards to specify. Local use may be made of higher font numbers.

@get_prop_len

`@get_prop_len` 0 must return 0. This is required by some Infocom games and files generated by old versions of Inform.

@set_cursor

S8.7.2.3 states that it is illegal to move the cursor outside the current size of the upper window. S8.8.3.5 gives the equivalent rule for Version 6.

Many modern games have been lax in obeying this rule; in particular some of the standard Inform menu libraries have violated it. Infocom's Sherlock also miscalculated the size of the upper window to use for box quotes.

It is recommended that if the cursor is moved below the split position in V4/V5, interpreters should execute an implicit "`split_window`" to contain the requested cursor position, if possible. Diagnostics should be produced, but should be suppressable.

In V6, it is legal to position the cursor up against the right or bottom of a window - eg at (1,1) in a zero-sized window or at (641,401) in 640x400 window. Indeed, this is the default state of windows 1 to 7, and the cursor may be left at the right-hand side of a window when wrapping is off.

Attempting to print text (including new-lines) when the cursor is fewer than `font_height` units from the bottom of the window results in undefined behaviour - this precludes any printing in windows less than `font_height` units high.

@split_window

In Version 6, the Standard states that the cursor remains in the same absolute position. This is an extension to the Z-machine, as this rule is not followed by Infocom's own interpreters, which

keep the same window-relative position. The Standard behaviour more closely mimics Version 5's `split_window`.

The opcode dictionary incorrectly states that "the width and x-coordinates of windows 0 and 1 are not altered." The correct behaviour is detailed in S8.8.4.1.

`@output_stream`

Interpreter authors are reminded that "nested" uses of `@output_stream` 3 have been required since Standard 1.0; see S7.1.2.1.1.

`@read_mouse`

`@read_mouse` is realtime. When called it must read the current mouse location, whether or not the mouse is inside the current mouse window. Most modern interpreters get this wrong, but Infocom's interpreters behave this way. Interpreters are allowed to show positions and button states outside the Z-machine screen if the pointer is outside the interpreter's own user interface (using negative values if needed).

Programs must be prepared to cope with this. For example in a painting program you might want to ignore all buttons down outside the screen. When dragging something you might want to keep trying to follow the pointer, even outside the screen, until the buttons are released.

Interpreters may constrain the pointer to the screen as long as buttons are held down - this might aid dragging operations - although this is not required.

Previous revisions of the Standard stated that mouse buttons were numbered in the following fashion: "low bits on the right of the mouse, rising as one looks left". This is a rather non-portable implementation and should be abandoned in modern interpreters in favour of the "primary" button being the lowest bit, the "secondary" (if present) being the next lowest bit, and so on, up to a potential 16 buttons. The ordering of buttons should be that which is most natural for the host system. Here are some suggested assignments:

Button assignments			
Platform	**Bit 0 (low)**	**Bit 1**	**Bit 2**
RISC OS	Select	Adjust	Menu
MacOS	Primary/only	Secondary	Tertiary
Windows	Left	Right	Middle
X	Left	Right	Middle

There is no way for a story file to ascertain how many buttons are available to the user, so it is recommended that vital functions are not accessible only via secondary mouse buttons. Some of a platform's buttons may be reserved for the interpreter's use and not visible to the story file, in which case the remaining buttons must be packed down to the lowest bits.

Mouse clicks

In Version 5, mouse clicks are always represented with an input ZSCII code of 254.

In Version 6, a single click, or the first click of a double-click, is passed in as 254. The second click of a double-click is passed in as 253.

`@picture_data`

`@picture_data` 0 branches if any pictures are available.

`@sound_effect`

To clarify:

```
@sound_effect number 3/4
```

will stop (and optionally unload) sound "number" if it is currently playing (or loaded). Otherwise it is ignored.

```
@sound_effect 0 3/4
```

will stop (and unload) all sounds - music and effects.

The "repeats" parameter in Version 5 indicates the total number of times to play the sound, not the number of times to repeat it after the first play. Setting repeats to zero in V5 is illegal - it is suggested that interpreters treat this as a request to play the sound once, and maybe issue a warning.

Callback routines are only called when the sound has played the requested number of times. If manually stopped or interrupted by another sound, the routine is not called.

The Blorb specification effectively revised Standard 1.0, as follows:

"Sound" is split into two classes - "music" (eg Blorb SONG and MOD) and "effects" (eg Infocom .SND or Blorb AIFF).

The "sound requested" header bit is left set if the interpreter supports either effects or music. It is only cleared if the interpreter can provide neither effects nor music.

Music and effects are treated as two separate channels. Playing a new effect interrupts the current effect, but not the music, and vice versa. Whether loading a new sound (with `@sound_effect n 1`) stops the current one is implementation defined.

As of Standard 1.1, there is a header bit to indicate that the game wants to use multiple sound channels and the interpreter can handle it, and a bit to indicate the availability of music.

Volume guidelines

[*This would better live in the Blorb specification, but is partly Z-machine specific.*]

When using Blorb resources, the default interpreter behaviour (unless over-ridden by the player) should be for samples played at maximum volume (64), in one channel of a SONG or MOD played at volume 8, to be of equal volume to samples played at maximum volume (8) as an effect. This will be the natural behaviour if effects use one physical channel and MODs/SONGs use four physical channels.

Ideally, a sound played at volume `<n>` in a SONG played at volume `<m>` should sound the same as when played as an effect at volume `<n*m/64>`. This mandates that the volume scale for effects be equivalent to the scale defined for samples in the MOD specification.

If multi-channel effects are used, the overall volume should be independent of the number of channels used in the sound. Thus a stereo AIFF containing the same samples for left and right

should sound as loud as a mono AIFF containing the same data. This will need adjustment of volume if stereo AIFFs use two physical channels and mono AIFFs use one. No adjustment would be required if an interpreter reduced all AIFFs to mono.

Colour numbers

The presence of the "under the cursor" colour option, and in Standard 1.1 the `set_true_colour` opcode (see below), raises the possibility of non-standard colour numbers appearing in window property 11. In addition, an interpreter may offer non-standard default colours, which need to appear in the story file header. We now give more detailed implementation guidelines.

If a true colour, or an "under the cursor" colour, has been requested by the game, then the foreground or background colour shown in window property 11 is implementation defined, except that:

1. If the colour selected was one of the standard set (2-15), then that colour is indicated in property 11.
2. If the colour selected was not one of the standard set, the colour shown in property 11 will be >= 16.

It is legal for interpreters to always show the same value in property 11 if a true or sampled colour is in use. As a result, story files cannot assume that setting a value that was read from property 11 will give the same colour, if `@set_true_colour` or `@set_colour -1` has been used in that window.

The same rules apply if an interpreter offers non-standard default colours (which need to be shown in the header) although in this case it would be ill-advised to show the same colour numbers for foreground and background - unless they can be distinguished, non-standard default colours should probably not be offered.

If the interpreter offers a limited palette, then there is no problem, as it can be arranged for there to be fewer than 240 distinct non-standard colours. In an interpreter with a higher colour-depth, a good implementation would be to use colours 16-255 to represent the last 240 distinct non-standard colours used, re-using numbers after 240 colours have been used. This will minimize potential problems caused by non-standard colours, particularly when set as defaults.

Regardless of the limitations on colour numbers, in Version 6 each window must remember accurately the colour pair selected, so it is preserved across window switches.

Additions

Blorb

Blorb is the standard resource format for the Z-machine. Games should be distributed as

a) a single story file;

b) a story file with accompanying Blorb file;

or c) a stand-alone executable Blorb file.

To minimise user confusion, Standard 1.1 interpreters should be able to run Z-code executables contained in either a plain story file, or embedded in a stand-alone Blorb file. If the interpreter supports sound or graphics, it should be able to obtain those resources from either a stand-alone or separate Blorb file (although the Standard does not preclude other formats).

Embedded interpreters, or interpreters running on small platforms, may accept either Z-code or resources in some custom format, but these should come with some form of conversion utility. In the minimum case of a text-only interpreter, this might take the form of a Blorb to story file converter.

Quetzal

Quetzal is the standard saved game format for the Z-machine. Standard 1.1 interpreters are recommended to save games in Quetzal format (by default), and to be able to load both compressed and uncompressed Quetzal files.

Header Extension

If the interpreter claims Standard 1.1 compliance then it supports the following extra words in the header extension table. If the header extension table is present, and the extra words are present, then a Standard 1.1 interpreter will examine and update them.

If the interpreter is not Standard 1.1, then the extra words will be ignored. Guidelines on behaviour in this case are supplied in the descriptions below.

Note that the header extension table itself is a Version 5 feature, so the extra information here is not available in earlier Versions, despite the fact that some fields could be applicable.

The new flags are placed in a new word, rather than in Flags 2, to minimize potential compatibility problems. To prevent future compatibility problems, all reserved bits in the Flags 3 word MUST be cleared by the interpreter.

Word	V	Dyn	Int	Rst	Contents
4	5				Flags 3:
	5		*	*	0: If set, game wants to use music
	5		*	*	1: If set, game wants multiple sound channels
	6		*	*	2: If set, game wants to use transparency
					(For all bits, Int clears again if it cannot provide the requested effect).
5	5		*	*	True default foreground colour
6	5		*	*	True default background colour

0: If set, game wants to use music

A game file that wishes to use music will set this bit. If an interpreter can provide music (as described in the Blorb specification) it must leave the bit set; if it is unable to do this, it must clear the bit. If this bit is set then a minimum of two sound channels exist, one for effects and one for music.

If this flags word is not present, or the interpreter predates Standard 1.1, then there is no way to tell whether music is available. If an interpreter doesn't support music, then `@sound_effect` opcodes requesting music will be ignored.

If a game that supplies this flags word does not set this bit, then the interpreter may choose to act as though music were unsupported.

1: If set, game wants multiple sound channels

If the game wishes to have multiple sound channels, it must set this bit. This changes the behaviour of `@sound_effect`. An interpreter must clear this bit if it can't play more than one sound of a given type a time.

Interpreters predating standard 1.1 will not support multiple sound channels, and will act as though this bit is clear.

The minimum requirement on an interpreter that sets this bit is that it can play at least two mono effects and one (4-channel) piece of music simultaneously. More channels may be available (see `@sound_effect` section below for details).

2: If set, game wants to use transparency

In Version 6 colour 15 is defined as transparent. If an interpreter cannot provide transparency then it must clear this bit.

Transparency is a new feature in Standard 1.1. Older interpreters will not support this.

All other bits in the flags word must be cleared by the interpreter.

Unicode in strings

In Standard 1.1, for Version 5 and later, a new mechanism for storing Unicode text in standard Z-machine strings is defined - the `@print_unicode` opcode in Standard 1.0 is not adequate to straightforwardly handle, say, a Japanese-language game, which would require widescale use of non-ZSCII characters.

When encountered in Z-machine text, ZSCII characters 768-1023 are "Unicode escapes", which indicate that Unicode data follows. The number of Unicode codepoints to follow is given by the ZSCII code minus 767.

After reading a Unicode escape, the Z-machine starts reading Unicode at the next word. Once the designated number of Unicode characters have been read, text decoding resumes where it left off in the word where the Unicode escape was found. Once that word has been processed, the next word is read from after the Unicode sequence, unless the end-bit was set. This ordering will arise naturally from typical decoder implementations, so is easier to decode than to explain.

The Unicode text is stored as a series of words, inverted to prevent casual browsing.

This would be a typical usage:

```
"Japanese" in Japanese: (U+65E5 U+672C U+8A9E)
Z-characters: 5,6,24, 2,5,5, <Unicode>, <Unicode>, <Unicode>
Encoded:  $10DC $88A5 $D15F $98D3 $75D1
```

The following examples demonstrate ordering:

```
"Zork<TM>" (<TM> = U+2122)
Z-characters: 4,31,20, 23,16,5, 6,24,0, <Unicode>
Encoded: $13F4 $5E05 $9B00 $DEDD

"Zor<TM>k"
Z-characters: 4,31,20, 23,5,6, 24,0,16, <Unicode>
Encoded: $13F4 $5CA6 $E010 $DEDD

"Zo<TM>rk"
Z-characters: 4,31,20, 5,6,24, 0,23,16, <Unicode>
Encoded: $13F4 $14D8 $82F0 $DEDD

"Z<TM>ork"
Z-characters: 4,31,5, 6,24,0, <Unicode>, 20,23,16
Encoded: $13E5 $1B00 $DEDD $D2F0
```

[Occasional characters like that `<TM>` should normally be output using `print_unicode` (with a Standard interpreter check) for backwards compatibility.]

ZSCII characters 768-1023 can only be encoded as a 4 Z-character sequence, leading to a basic overhead of 4 bytes on all pure Unicode strings.

Unicode escapes can only be used in strings, not `@print_char`, and are not used when tokenising text.

`@save` and `@restore`

```
EXT:0 0 5/* save table bytes name prompt -> (result)

EXT:1 1 5/* restore table bytes name prompt -> (result)
```

As of Standard 1.1 an additional optional parameter, prompt, is allowed on Version 5 extended save/restore. This allows a game author to tell the interpreter whether it should ask for confir-

mation of the provided file name (`prompt=1`), or just silently save/restore using the provided filename (`prompt=0`).

If the parameter is not provided, whether to prompt or not is a matter for the interpreter - this might be globally user-configurable. Infocom's interpreters do prompt for filenames, many modern ones do not.

`@buffer_screen`

New opcode in Standard 1.1, for Version 6

```
EXT:29 1D 6/* buffer_screen mode -> (result)
```

Interpreters may use a backing store to store the Z-machine screen state, rather than plotting directly to the screen. This would normally be the case in a windowed operating system environment. If a backing store is in use, display changes executed by the Z-machine may not be immediately made visible to the user. Standard 1.1 adds the new opcode `@buffer_screen` to Version 6 to control screen updates. An interpreter is free to ignore the opcode if it doesn't fit its display model (in which case it must act as if `buffer_screen` is always set to 0).

When `buffer_screen` is set to 0 (the default), all display changes are expected to become visible to the user either immediately, or within a short period of time, at the interpreter's discretion. At a minimum, all updates become visible before waiting for input. Any intermediate display states between input requests may not be seen; for example when printing a large amount of new text into a scrolling window, all the intermediate scroll positions may or may not be shown.

When `buffer_screen` is set to 1, the interpreter need not change the visible display at all. Any display changes can be done purely in the backing store. A program may set `buffer_screen` to 1 before carrying out a complex layered graphical composition, to indicate that the intermediate states are not worth showing. When `buffer_mode` is set back to 0, the display is not necessarily updated immediately - if this is required, `buffer_mode -1` must be issued as well. It would be extremely ill-advised to prompt for input with `@buffer_screen` set to 1.

With `buffer_screen` in either state, an update of the visible display can be forced immediately by issuing `@buffer_screen -1`, without altering the current buffering state. Note that `@buffer_screen -1` does NOT flush the text buffer.

`@buffer_screen` returns the old `buffer_screen` state.

`@set_text_style`

As of Standard 1.1, it is legal to request style combinations in a single `@set_text_style` opcode by adding the values (which are powers of two) together. If the parameter is non-zero, then all the styles given are activated. If the parameter is zero, then all styles are deactivated.

If the interpreter is unable to provide the requested style combination, it shall give precedence first to the styles requested in the most recent call to `@set_text_style`, and within that to the highest bit, making the priority Fixed, Italic, Bold, Reverse.

Although a story file can determine which individual styles are available by inspecting the header, there is no indication of which styles can be combined. To improve this situation, at least for Version 6, Standard 1.1 requires window property 10 to show the actual style combination currently in use; with this a story file can probe for the availability of particular combinations. Pre-existing interpreters may or may not already do this correctly.

@set_font

```
EXT:4 4 6/- set_font font window -> (result)
```

In Version 6, `set_font` has an optional window parameter, as for `set_colour`. This was part of the original Infocom design, but omitted by earlier Standards. It is reinstated here, as it is useful to be able to measure a font that is about to be used in another window, so that window can be sized before attempting to place the cursor in it.

@set_colour

In Version 6 only, colour 15 is defined as transparent. This is only valid as a background colour; an attempt to select it for the foreground should produce a diagnostic. Interpreters not supporting transparency shall ignore any attempt to select colour 15.

If the current background colour is transparent, then printed text is superimposed on the current window contents, without filling the background behind the text. `@erase_window`, `@erase_line` and `@erase_picture` become null operations.

The intent is to make it possible to superimpose text on non-uniform images. Up until now, only uniform images could be satisfactorily written on by sampling the background colour - that in itself would be problematical if the interpreter used dithering.

Scrolling with the background set to transparent is not permitted, so transparent should only be requested in a non-scrolling window. It is not valid to use Reverse Video style with the background set to transparent.

Instructions that prompt for user input, such as `@read` and `@save`, should be avoided when the background is set to transparent, as it will not generally be possible for text entry to take place satisfactorily in the absence of a defined background colour.

Printing text multiple times on top itself with the background set to transparent should be avoided, as the interpreter may use anti-aliasing, resulting in the text getting progressively heavier.

Standard 1.1 interpreters must provide the following basic colour set, regardless of machine type, in both Version 5 and Version 6:

```
-1 = sample      (true -3) [V6 only]
 0 = current     (true -2)
 1 = default     (true -1)
 2 = black       (true $0000, $$0000000000000000)
 3 = red         (true $001F, $$0000000000011111)
 4 = green       (true $03E0, $$0000001111100000)
 5 = yellow      (true $03FF, $$0000001111111111)
 6 = blue        (true $7C00, $$0111110000000000)
 7 = magenta     (true $7C1F, $$0111110000011111)
 8 = cyan        (true $7FE0, $$0111111111100000)
 9 = white       (true $7FFF, $$0111111111111111)
10 = light grey  (true $5AD6, $$0101101011010110)
11 = medium grey (true $4631, $$0100011000110001)
12 = dark grey   (true $2D6B, $$0010110101101011)
13  reserved
14  reserved
15 = transparent (true -4) [V6 only]
```

This is the original Amiga Version 6 colour set. The greys are gamma adjusted from the original values (8-bit &AA,&77,&44) used on the Amiga, on the assumption that the Amiga's system gamma is 1/1.8.

The equivalences between the colour numbers and true colours are canonical - this gives an exact equivalence mapping from `@set_colour` to `@set_true_colour`.

Interpreters may provide different colours (eg making colour 10 dark grey), but if and only if they can detect they are running an original Infocom story file.

Standard 1.1 interpreters should support multiple text colours on-screen simultaneously, and not implement the Amiga-style behaviour described in Standard 1.0 (changing all text already on screen to the current colour). If the multiple-colour model is in use, the interpreter number header byte must not be set to "Amiga" (4), except by explicit user action, as this has historically been examined to determine the colour model in use.

When running story files in which the Flags 3 word is present, the multiple-colour model must be used.

`@set_true_colour`

New opcode in Standard 1.1, for Version 5 or later, if Flags 1 bit 0 is set.

```
EXT:13 D 5/* @set_true_colour foreground background
         6/* @set_true_colour foreground background window
```

fg and bg are 15-bit colour values - bit 15 = 0
bits 14-10 blue
bits 9-5 green
bits 4-0 red

Magic values in the opcode are: (-1) = default setting
(-2) = current setting
(-3) = colour under cursor (V6 only)
(-4) = transparent (V6 only)

The interpreter selects the closest approximations available to the requested colours. In V6, the interpreter may store the approximations in window properties 16 and 17, so the program can tell how close it got (although it is acceptable for the interpreter to just store the requested value).

In the minimal implementation, interpreters just need to match to the closest of the standard colours and internally call `@set_colour` (although that would have to ensure window properties 16 and 17 were updated). In a full implementation this would be turned around and `@set_colour` would internally call `@set_true_colour`.

The default true colours are stored in the header extension table.

The optional window parameter is only allowed in V6, and operates the same as in `@set_colour`.

True colour specifications are in the sRGB colour space, $0000 being black and $7FFF being white. Colours should be gamma adjusted if necessary. See the PNG specification for a good introduction to colour spaces and gamma correction.

`@get_wind_prop`

With the addition of true colours, two new Version 6 window properties are defined.

The two new window properties are:

```
16: true foreground colour
17: true background colour
```

These are not writable with `@put_wind_prop`.

The true foreground and background colours show the actual colour being used for the foreground and background, whether it was set using `@set_colour` or `@set_true_colour`. Transparent is indicated as -4. If the colour was sampled from a picture then the value shown may be a 15-bit rounding of a more precise colour, leading to a slight inaccuracy if the colour is read and then written back.

`@sound_data`

New opcode in Standard 1.1, for Version 5 or later.

```
EXT:14 E 5/* sound_data sound-number array ?(label)
```

Asks the interpreter for data on the sound with the given number. If the sound number is valid, and the sound is of a type supported by the intepreter, a branch occurs and information is written to the array: the type (1 for effect, 2 for music) in word 0, playing status in word 1, and channel availability in word 2. (This is an array, not a "table" with initial size information.)

Otherwise, if the sound number is zero, the interpreter writes the number of available (and supported) sounds into word 0 of the array and the release number of the sounds file (eg from the Blorb RelN chunk) into word 1. A branch occurs if the number of sounds is non-zero.

Otherwise, nothing happens.

The playing status is -1 if the sound is not currently playing. If the sound is currently playing, the playing status is the length of time that the sound has been playing, in tenths of a second, clamped to a maximum of 3276.7 seconds. If the sound is playing on multiple channels, it is the longest running channel's playing time that is indicated.

The channel availability word is set to 0 if playing the sound will not result in any other sounds being stopped. Otherwise the availability word contains the number of a sound that will be interrupted by playing this sound.

This opcode must be provided even if sound isn't supported (in which case it would always do nothing).

`@sound_effect`

Standard 1.1 adds new behaviour to `@sound_effect` (but see the clarifications section above for notes on pre-1.1 behaviour).

As of Standard 1.1, there is a header bit to indicate that the game wants to use multiple sound channels and the interpreter can handle it.

If the multiple sound channels bit is clear (or not present), then the interpreter behaves as per Standard 1.0 + Blorb 1.1 - see above for clarifications.

If the multiple sound channels bit is set, then the interpreter does not stop any current sound when a new one is played or loaded - it only stops a sound when explicitly asked to, or when it has played the specified number of repeats. That way, a game can play multiple sounds of the same type at once.

However, there are an undefined number of sound channels available. When an interpreter runs out of channels of a particular type, and an additional call to `@sound_effect` is made, the sound with the highest resource number of the same type is stopped (without its callback routine being called) and the new sound is played. It may be that playing a sound interrupts more than one sound - for example a stereo effect could interrupt two mono effects.

Sounds of different types can never interrupt each other - the channels used for music and effects are independent. As a corollary to this, it may be that more effects channels are available if the game does not ask to use music.

The highest resource number rule allows an author to indicate the "importance" of each sound.

The same sound can be played on more than one channel, with different volumes, repeats and callback routines. In the event of channel exhaustion, it is the "oldest" play of the sound with the highest resource number that gets stopped first.

`@sound_effect` <n> 3 (or 4) stops all plays of sound <n>.

The original Standard 1.0 + Blorb 1.1 behaviour is equivalent to the new scheme with one channel each of effects and music.

New opcode summary					
St	**Br**	**Opcode**	**Hex**	**V**	**Inform name and syntax**
*		EXT:0	0	5/*	save table bytes name prompt -> (result)
*		EXT:1	1	5/*	restore table bytes name prompt -> (result)
*		EXT:4	4	6/–	set font font window -> (result)
		EXT:13	D	5/*	set true colour foreground background
				6/*	set true colour foreground background window
	*	EXT:14	E	5/*	sound data sound-number array ?(label)
*		EXT:29	1D	6/*	buffer_screen mode -> (result)

`set_true_colour` is not a good candidate for taking a 2OP encoding like `set_colour`, as it will have a low usage frequency, and it will rarely take small constants.

This document was compiled and converted to PDF on 2002-06-16 by Peer Schaefer.
Bug reports, suggestions etc. to peerschaefer@gmx.net