Chapter VII: The Z-Machine

§41 Architecture and assembly

Infocom's games of 1979–89 were written in a language called ZIL, the Zork Implementation Language. At first sight this is outlandishly unlike Inform, but appearances are deceptive. The following source code describes toy boats in Kensington Park, from a game widely considered a masterpiece: 'Trinity' (1986), by Brian Moriarty.

```
<OBJECT BOAT
        (LOC ROUND-POND)
        (DESC "toy boats")
        (FLAGS TRYTAKE NODESC PLURAL)
        (SYNONYM BOAT BOATS TOYS)
        (ADJECTIVE TOY)
        (ACTION BOAT-F)>
<ROUTINE BOAT-F ()
         <COND (<VERB? EXAMINE WATCH>
                <TELL CTHEO
" are crafted of paper and sticks. They bob freely among the "
D ,POND-BIRDS ", who can barely conceal their outrage." CR>
                <RTRUE>)
               (<VERB? SWIM DIVE WALK-TO FOLLOW SIT LIE-DOWN ENTER>
                <DO-WALK ,P?IN>
                <RTRUE>)
               (<INTBL? ,PRSA ,TOUCHVERBS ,NTOUCHES>
                <TELL CTHE ,BOAT " are far out of reach." CR>
                <RTRUE>)
               (T
                <RFALSE>)>>
```

Inform and ZIL each have objects, with properties and attributes. They're called different things, true: ZIL has its dictionary words in `SYNONYM` and `ADJECTIVE` where Inform uses `name`, ZIL calls attributes "flags" and has `NODESC` where Inform would have `scenery`, but the similarity is striking. Both languages have routines with a tendency to return `true` and `false`, too.

The underlying similarity is the Z-machine, which both languages make use of. The Z-machine is an imaginary computer: created on Joel Berez's mother's coffee table in Pittsburgh in 1979, it has never existed as circuitry.

Instead, almost every real computer built in the 1980s and 1990s has been taught to emulate the Z-machine, and so to run story files.

This chapter contains what the advanced Inform programmer needs, from time to time, to know about the Z-machine. The reader who only wants to get at fairly easy screen effects like coloured text may want to turn straight to the references to §42, where a number of convenient library extensions are listed.

In any case this chapter is by no means the full story, which is contained in *The Z-Machine Standards Document*. It seems nonetheless appropriate to acknowledge here the work of the Z-machine's architects: Joel Berez, Marc Blank, P. David Lebling, Tim Anderson and others.

. . . . .

The Z-machine as conceived in 1979 is now known as "version 1", and there have been seven subsequent versions to date. Inform normally produces version 5 story files, but this can be controlled with the `-v` switch: so `-v6` compiles a version 6 story file, for instance. Briefly, the versions are:

Versions 1 and 2. Early draft designs by Infocom, used only in the first release of the 'Zork' trilogy. Inform cannot produce them.

Version 3. The standard Infocom design, limited in various ways: to 255 objects, 32 attributes, at most 4 entries in a property array, at most 3 arguments in a routine call and a story file at most 128K in size. Inform can produce version 3 files but this is not recommended and some advanced features of the language, such as message-sending, will not work.

Version 4. A partial upgrade, now seldom used.

Version 5. The advanced Infocom design and the one normally used by Inform. Limits are raised to 65,535 objects, 48 attributes, 32 entries in a property array, 7 arguments in a routine call and a story file at most 256K in size.

Version 6. Similar, though not identical, in architecture to Version 5, but offering support for pictures. Inform will compile to this, but there are two further obstructions: you need a way to package up the sounds and images to go with the story file (see §43), and then players need an interpreter able to make use of them.

Version 7. An intermediate version which has never proved useful, and whose use is now deprecated.

Version 8. Identical to version 5 except that it allows story files up to 512K long. Most of the largest Inform games use version 8.

. . . . .

The native language of the Z-machine is neither Inform nor ZIL, but an intermediate-level code which we'll call "assembly language". It's tiresome to write a program of any complexity in assembly language. For instance, here are two equivalent pieces of Inform: first, a statement in ordinary code:

```
"The answer is ", 3*subtotal + 1;
```

Secondly, assembly language which achieves the same end:

```
@print "The answer is ";
@mul 3 subtotal -> x;
@add x 1 -> x;
@print_num x;
@new_line;
@rtrue;
```

(Here we've used a variable called `x`.) Inform allows you to mix assembly language and ordinary Inform source code freely, but all commands in assembly language, called "opcodes", are written with an `@` sign in front, to distinguish them. The values supplied to opcodes, such as `3` and `subtotal`, are called "operands". The `->` arrow sign is read "store to" and indicates that an answer is being stored somewhere. So, for instance, the line

```
@add x 1 -> x;
```

adds `x` and 1, storing the result of this addition back in `x`. Operands can only be constants or variables: so you can't write a compound expression like `my_array-->(d*4)`.

As can be seen above, some opcodes store values and some don't. Another important category are the "branch" opcodes, which result in execution jumping to a different place in the source code if some condition turns out to be true, and not otherwise. For instance:

```
@je x 1 ?Xisone;
@print "x isn't equal to 1.";
.Xisone;
```

Here, `Xisone` is the name of a label, marking a point in the source code which a branch opcode (or an Inform `jump` statement) might want to jump to. (A label can't be the last thing in a function, but if you needed this, you could always finish with a label plus a `return` statement instead.) `@je` means "jump

if equal'', so the code tests `x` to see if it's equal to 1 and jumps to `Xisone` if so. Inform will only allow branches to labels in the same routine. Note that inserting a tilde,

```
@je x 1 ?~Xisntone;
```

reverses the condition, so this opcode branches if `x` is not equal to 1.

The full specification of Inform's assembly-language syntax is given in §14 of *The Z-Machine Standards Document*, but this will seldom if ever be needed, because the present chapter contains everything that can't be done more easily without assembly language anyway.

. . . . .

△ The rest of this section sketches the architecture of the Z-machine, which many designers won't need to know about. Briefly, it contains memory in three sections: readable and writeable memory at byte addresses 0 up to $S-1$, read-only memory from S up to $P-1$ and inaccessible memory from P upwards. (In any story file, the Inform expression `0-->2` gives the value of P and `0-->7` gives S.) The read-write area contains everything that needs to change in play: variables, object properties and attributes, arrays and certain other tables; except for the stack and the ''program counter'', its marker as to which part of some routine it is currently running. The beginning of the read-write area is a 64-byte ''header''. Byte 0 of this header, and so of an entire story file, contains the version number of the Z-machine for which it is written. (The expression `0->0` evaluates to this.)

The read-only area contains tables which the Inform parser needs to make detailed use of but never alters: the grammar lines and the dictionary, for instance. The ''inaccessible'' area contains routines and static (that is, unalterable) strings of text. These can be called or printed out, which is access of a sort, but you can't write code which will examine them one byte at a time.

In addition to local and global variables, the Z-machine contains a ''stack'', which is accessed with the name `sp` for ''stack pointer''. The stack is a pile of values. Each time `sp` is written to, a new value is placed on top of the pile. Each time it is read, the value being read is taken off the pile. At the start of a routine, the stack is always empty.

There is no access to the outside world except by using certain opcodes. For instance, `@read` and `@read_char` allow use of the keyboard, whereas `@print` and `@draw_picture` allow use of the screen. (The screen's image is not stored anywhere in memory, and nor is the state of the keyboard.) 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 story file 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.

§42 Devices and opcodes

This section covers the only opcodes which designers are likely to have occasional need of: those which drive powerful and otherwise inaccessible features of the Z-machine's "hardware", such as sound, graphics, menus and the mouse. There's no need to be fluent in assembly language to use these opcodes, which work just as well if used as incantations from a unfamiliar tongue.

• **WARNING**

Some of these incantations may not work well if a story file is played on old interpreters which do not adhere to the Z-Machine Standard. Standard interpreters are very widely available, but if seriously worried you can test in an `Initialise` routine whether your game is running on a good interpreter, as in the following code.

```
if (standard_interpreter == 0) {
    print "This game must be played on an interpreter obeying the
           Z-Machine Standard.^";
    @quit;
}
```

The library variable `standard_interpreter` holds the version number of the standard obeyed, with the upper byte holding the major and the lower byte the minor version number, or else zero if the interpreter isn't standard-compliant. Thus `$002` means 0.2 and `$100` means 1.0. Any standard interpreter will carry out the opcodes in this chapter correctly, or else provide fair warning that they cannot. (For instance, an interpreter running on a palm-top personal organiser without a loudspeaker cannot provide sound effects.) Here is how to tell whether a standard interpreter can or can't provide the feature you need.

Feature	*Versions*	*Available* `if`
auxiliary files	5,6,8	`(true)`
coloured text	5,6,8	`((0->1) & 1 ~= 0)`
input streams	5,6,8	`(true)`
menus	6	`(($10-->0) & 256 ~= 0)`
mouse	5,6	`(($10-->0) & 32 ~= 0)`
output streams	5,6,8	`(true)`
pictures	6	`(($10-->0) & 8 ~= 0)`
sounds	5,6,8	`(($10-->0) & 128 ~= 0)`
throw/catch stack frames	5,6,8	`(true)`
timed keyboard interrupts	5,6,8	`((0->1) & 128 ~= 0)`

For instance, if coloured text is essential (for instance if red and black letters have to look different because it's a vital clue to some puzzle), you may want to add a test like the following to your `Initialise` routine:

```
if ((0->1) & 1 == 0)
    print "*** This game is best appreciated on an interpreter
           capable of displaying colours, unlike the present
           one. Proceed at your own risk! ***^";
```

. . . . .

△ Text flows in and out of the Z-machine continuously: the player's commands flow in, responses flow out. Commands can come in from two different "input streams", only one of which is selected at any given time: stream 0 is the keyboard and stream 1 is a file on the host computer. The stream is selected with:

```
@input_stream number
```

The Inform debugging verb "replay" basically does no more than switch input to stream 1.

△ There are four output streams for text, numbered 1 to 4. These are: (1) the screen, (2) the transcript file, (3) an array in memory and (4) a file of commands on the host computer. These can be active in any combination, except that at all times either stream 1 or stream 3 is active and not both. Inform uses stream 3 when the message `print_to_array` is sent to a string, and streams 2 and 4 in response to commands typed by the player: "script on" switches stream 2 on, "script on" switches it off; "recording on" and "off" switch stream 4 on and off. The relevant opcode is:

```
@output_stream number arr
```

If `number` is 0 this does nothing. $+n$ switches stream n on, $-n$ switches it off. The `arr` operand is omitted except for stream 3, when it's a `table` array holding the text printed: that is, `arr-->0` contains the number of characters printed and the text printed is stored as ZSCII characters in `arr->2`, `arr->3`, ...

△ As the designer, you cannot choose the filename of the file of commands used by input stream 1 or output stream 4. Whoever is playing the story file will choose this: perhaps after being prompted by the interpreter, perhaps through a configuration setting on that interpreter.

• △△**EXERCISE 122**

Implement an Inform version of the standard 'C' routine `printf`, taking the form

```
printf(format, arg1, ...)
```

to print out the format string but with escape sequences like %d replaced by the arguments (printed in various ways). For example,

```
printf("The score is %e out of %e.", score, MAX_SCORE);
```

should print something like "The score is five out of ten."

△ In Version 6 story files, only, @output_stream can take an optional third operand when output stream 3 is being enabled. That is:

```
@output_stream 3 arr width
```

If width is positive, the text streamed into the array will be word-wrapped as if it were on a screen width characters wide; if width is negative, then as if on a screen -width pixels wide. The text going into arr is in the form of a sequence of lines, each consisting of a word containing the number of characters and then the ZSCII characters themselves in bytes. The sequence of lines finishes with a zero word. Such an array is exactly what is printed out by the opcode @print_form arr.

. . . . .

△ The Z-machine has two kinds of "screen model", or specification for what can and can't be done to the contents of the screen. Version 6 has an advanced graphical model, whereas other versions have much simpler textual arrangements. Early versions of the Z-machine are generally less capable here, so this section will document only the Version 5 and Version 6 models. (Versions 7 and 8 have the same model as Version 5.)

The version 5 screen model. The screen is divided into an upper window, normally used for a status line and sometimes also for quotations or menus, and a lower window, used for ordinary text. At any given time the upper window has a height H, which is a whole number of lines: and H can be zero, making the upper window invisible. (The story file can vary H from time to time and many do.) When text in the upper and lower windows occupy the same screen area, it's the upper window text that's visible. This often happens when quotation boxes are displayed.

```
@split_window H
```

Splits off an upper-level window of the given number of lines H in height from the main screen. Be warned that the upper window doesn't scroll, so you need to make H large enough for all the text you need to fit at once.

```
@set_window window
```

Selects which window text is to be printed into: (0) the lower one or (1) the upper one. Printing on the upper window overlies printing on the lower, is always done in a fixed-pitch font and does not appear in a printed transcript of the game.

```
@set_cursor line column
```

Places the cursor inside the upper window, where $(1, 1)$ is the top left character.

`@buffer_mode flag`

This turns on (`flag==true`) or off (`flag==false`) word-breaking for the current window: that is, the practice of printing new-lines only at the ends of words, so that text is neatly formatted.

`@erase_window window`

Blanks out window 0 (lower), window 1 (upper) or the whole screen (if `window=-1`).

Using fixed-pitch measurements, the screen has dimensions X characters across by Y characters down, where X and Y are stored in bytes `$21` and `$20` of the header respectively. It's sometimes useful to know this when formatting tables:

```
print "My screen has ", 0->$20, " rows and ", 0->$21, " columns.^";
```

Be warned: it might be 80×210 or then again it might be 7×40. Text printing has a given foreground and background colour at all times. The standard stock of colours is:

0	*current colour*	5	yellow
1	*default colour*	6	blue
2	black	7	magenta
3	red	8	cyan
4	green	9	white

`@set_colour foreground background`

If coloured text is available, this opcode sets text to be `foreground` against `background`. (But bear in mind that not all interpreters can display coloured text, and not all players enjoy reading it.) Even in a monochrome game, text can be set to print in "reverse colours": background on foreground rather than vice versa. Status lines are almost always printed in reverse-colour, but this is only a convention and is not required by the Z-machine. Reverse is one of five possible text styles: roman, bold, underline (which many interpreters will render with italic), reverse and fixed-pitch. (Inform's `style` statement chooses between these.)

• △**EXERCISE 123**
Design a title page for 'Ruins', displaying a more or less apposite quotation and waiting for a key to be pressed. (For this last part, see below.)

• △**EXERCISE 124**
Change the status line so that it has the usual score/moves appearance except when a variable `invisible_status` is set to `true`, when it's invisible.

• △**EXERCISE 125**
Alter the 'Advent' example game to display the number of treasures found instead of the score and turns on the status line.

• △**EXERCISE 126**
(From code by Joachim Baumann.) Put a compass rose on the status line, displaying the directions in which the room can be left.

• △△**EXERCISE 127**
(Cf. 'Trinity'.) Make the status line consist only of the name of the current location, centred in the top line of the screen.

The version 6 screen model. We are now in the realm of graphics, and the screen is considered to be a grid of pixels: coordinates are usually given in the form (y, x), with $(1, 1)$ at the top left. y and x are measured in units known, helpfully enough, as "units". The interpreter decides how large "1 unit" is, and it's not safe to assume that 1 unit equals 1 pixel. All you can tell is what the screen dimensions are, in units:

```
print "The screen measures ", $22-->0, " units across and ",
      $22-->1, " units down.^";
```

There are eight windows, numbered 0 to 7, which text and pictures can currently be printing to: what actually appears on the screen is whatever shows through the boundaries of the window at the time the printing or plotting happens. Window number -3 means "the current one". Windows have no visible borders and usually lie on top of each other. 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". If you move a window so that the cursor is left outside, the interpreter automatically moves the cursor back to the window's new top left. If you only move the cursor, it's your responsibility to make sure it doesn't leave the window.

The attributes are (0) "wrapping", (1) "scrolling", (2) "copy text to output stream 2 if active" and (3) "buffer printing". Wrapping means that when text reaches the right-hand edge it continues from the left of the next line down. Scrolling means scrolling the window upwards when text printing reaches the bottom right corner, to make room for more. Output stream 2 is the transcript file, so the question here is whether you want text in the given window to appear in a transcript: for instance, for a status line the answer is probably "no", but for normal conversation it would be "yes". Finally, buffering is a more sophisticated form of wrapping, which breaks lines of text in between words, but which (roughly speaking) means that no line is printed until complete. Note that ordinary printing in the lower window has all four of these attributes.

```
@window_style window attrs operation
```

Changes window attributes. `attrs` is a bitmap in which bit 0 means "wrapping", bit 1 means "scrolling", etc. `operation` is 0 to set to these settings, 1 to set only those attributes which you specify in the bitmap, 2 to clear only those and 3 to reverse them. For instance,

```
@window_style 2 $$1011 0
```

sets window 2 to have wrapping, scrolling and buffering but not to be copied to output stream 2, and

```
@window_style 1 $$1000 2
```

clears the buffer printing attribute of window 1.

Windows have 16 properties, numbered as follows:

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

The x and y values are all in units, but the margin sizes are in pixels. The font size data is `256*h+w`, where `h` is the height and `w` the width in pixels. The colour data is `256*b+f`, where `f` and `b` are foreground and background colour numbers. The text style is a bitmap set by the Inform `style` statement: bit 0 means Roman, 1 is reverse video, 2 is bold, 3 is italic, 4 is fixed-pitch. The current value of any property can be read with:

```
@get_wind_prop window prop -> r
```

Those few window properties which are not italicised in the table (and only those few) can be set using:

```
@put_wind_prop window prop value
```

Most window properties, the ones with italicised text in the table above, are set using specially-provided opcodes:

```
@move_window window y x
```

Moves to the given position on screen. Nothing visible happens, but all future plotting to the given window will happen in the new place.

```
@window_size window y x
```

Changes window size in pixels. Again, nothing visible happens.

```
@set_colour foreground background window
```

Sets the foreground and background colours for the given window.

`@set_cursor line column window`

Moves the window's cursor to this position, in units, relative to $(1, 1)$ in the top left of the window. If this would lie outside the margin positions, the cursor moves to the left margin of its current line. In addition, `@set_cursor -1` turns the cursor off, and `@set_cursor -2` turns it back on again.

`@get_cursor arr`

Writes the cursor row of the current window into `arr-->0` and the column into `arr-->1`, in units.

`@set_font font -> r`

(This opcode is available in Versions 5 and 8 as well as 6.) Selects the numbered font for the current window, and stores a positive number into `r` (actually, the previous font number) if available, or zero if not available. Font support is only minimal, for the sake of portability. The possible font numbers are: 1 (the normal font), 3 (a character graphics font: see §16 of *The Z-Machine Standards Document*), 4 (a fixed-pitch Courier-like font). Owing to a historical accident there is no font 2.

`@set_margins left right window`

Sets margin widths, in pixels, for the given window. If the cursor is overtaken in the process and now lies outside these margins, it is moved back to the left margin of the current line.

△ "Interrupt countdowns" are a fancy system to allow text to flow gracefully around obstructions such as pictures. If property 9 is set to a non-zero value, then it'll be decremented on each new-line, and when it hits zero the routine in property 8 will be called. This routine should not attempt to print any text, and is usually used to change margin settings.

• **EXERCISE 128**
(Version 6 games only.) Set up wavy margins, which advance inwards for a while and then back outwards, over and over, so that the game's text ends up looking like a concertina.

Here are two useful tricks with windows:

`@erase_window window`

Erases the window's whole area to its background colour.

`@scroll_window window pixels`

Scrolls the given window by the given number of pixels. A negative value scrolls backwards, i.e., moving the body of the window down rather than up. Blank (background colour) pixels are plotted onto the new lines. This can be done to any window and is not related to the "scrolling" attribute of a window.

△ Finally, Version 6 story files (but no others) are normally able to display images, or "pictures". To the Z-machine these are referred to only by number, and a story file does not "know" how pictures are provided to it, or in what format they are stored. For the mechanics of how to attach resources like pictures to a story file, see §43. Four opcodes are concerned with pictures:

```
@draw_picture pn y x
```

Draws picture number `pn` so that its top left corner appears at coordinates (`y`, `x`) in units on the screen. If `y` or `x` are omitted, the coordinate of the cursor in the current window is used.

```
@erase_picture pn y x
```

Exactly as `@draw_picture`, but erases the corresponding screen area to the current background colour.

```
@picture_data pn arr ?Label
```

Asks for information about picture number `pn`. If the given picture exists, a branch occurs to the given `Label`, and the height and width in pixels of the image are written to the given array `arr`, with `arr-->0` being the height and `arr-->1` the width. If the given picture doesn't exist, no branch occurs and nothing is written, *except* that if `pn` is zero, then `arr-->0` is the number of pictures available to the story file and `arr-->1` the "release number" of the collection of pictures. (Or of the Blorb file attached to the story file, if that's how pictures have been provided to it.)

```
@picture_table tarr
```

Given a `table` array `tarr` of picture numbers, this warns the Z-machine that the story file will want to plot these pictures often, soon and in quick succession. Providing such a warning is optional and enables some interpreters to plot more quickly, because they can cache images in memory somewhere.

. . . . .

△ Sound effects are available to story files of any Version from 5 upwards. Once again, to the Z-machine these are referred to only by number, and a story file does not "know" how sounds are provided to it, or in what format they are stored. For the mechanics of how to attach resources like sound effects to a story file, see §43. There is only one sound opcode, but it does a good deal. The simplest form is:

```
@sound_effect number
```

which emits a high-pitched bleep if `number` is 1 and a low-pitched bleep if 2. No other values are allowed.

```
@sound_effect number effect volrep routine
```

The given `effect` happens to the given sound `number`, which must be 3 or higher and correspond to a sound effect provided to the story file by the designer. Volume is measured from 1 (quiet) to 8 (loud), with the special value 255 meaning "loudest possible", and you can also specify between 0 and 254 repeats, or 255 to mean "repeat forever". These two parameters are combined in `volrep`:

```
volrep = 256*repeats + volume;
```

The `effect` can be: 1 (prepare), 2 (start), 3 (stop), 4 (finish with). You may want to "warn" the Z-machine that a sound effect will soon be needed by using the "prepare" effect, but this is optional: similarly you may want to warn it that you've finished with the sound effect for the time being, but this too is optional. "Start" and "stop" are self-explanatory except to say that sound effects can be playing in the background while the player gets on with play: i.e., the Z-machine doesn't simply halt until the sound is complete. The "stop" effect makes the sound cease at once, even if there is more still to be played. Otherwise, unless set to repeat indefinitely, it will end by itself in due course. If a `routine` has been provided (this operand is optional, and takes effect only on `effect` 2), this routine will then be called as an interrupt. Such routines generally do something like play the sound again but at a different volume level, giving a fading-away effect.

. . . . .

△ In addition to reading entire lines of text from the keyboard, which games normally do once per turn, you can read a single press of a key. Moreover, on most interpreters you can set either kind of keyboard-reading to wait for at most a certain time before giving up.

```
@aread text parse time function -> result
```

This opcode reads a line of text from the keyboard, writing it into the `text` string array and 'tokenising' it into a word stream, with details stored in the `parse` string array (unless this is zero, in which case no tokenisation happens). (See §2.5 for the format of `text` and `parse`.) While it is doing this it calls `function()` every `time` tenths of a second: the process ends if ever this function returns true. The value written into `result` is the "terminating character" which finished the input, or else 0 if a time-out ended the input.

```
@read_char 1 time function -> result
```

Results in the ZSCII value of a single keypress. Once again, `function(time)` is called every `time` tenths of a second and may stop this process early. (The first operand is always 1, meaning "from the keyboard".)

`@tokenise text parse dictionary`

This takes the text in the `text` buffer (in the format produced by `@aread`) and tokenises it, i.e., breaks it up into words and finds their addresses in the given `dictionary`. The result is written into the `parse` buffer in the usual way.

`@encode_text zscii-text length from coded-text`

Translates a ZSCII word to the internal, Z-encoded, text format suitable for use in a `@tokenise` dictionary. The text begins at `from` in the `zscii-text` and is `length` characters long, which should contain the right length value (though in fact the interpreter translates the word as far as a zero terminator). The result is 6 bytes long and usually represents between 1 and 9 letters.

It's also possible to specify which ZSCII character codes are ''terminating characters'', meaning that they terminate a line of input. Normally, the return key is the only terminating character, but others can be added, and this is how games like 'Beyond Zork' make function keys act as shorthand commands. For instance, the following directive makes ZSCII 132 (cursor right) and 136 (function key f4) terminating:

```
Zcharacter terminating 132 136;
```

The legal values to include are those for the cursor, function and keypad keys, plus mouse and menu clicks (see Table 2 for values). The special value 255 makes all of these characters terminating. (For other uses of `Zcharacter`, see §36.)

• **EXERCISE 129**
Write a ''press any key to continue'' routine.

• **EXERCISE 130**
And another routine which determines if any key is being held down, returning either its ZSCII code or zero to indicate that no key is being held down.

• **EXERCISE 131**
Write a game in which a player taking more than ten seconds to consider a command is hurried along.

• **EXERCISE 132**
And if thirty seconds are taken, make the player's mind up for her.

• **EXERCISE 133**
Design an hourglass fixed to a pivot on one room's wall, which (when turned upright) runs sand through in real time, turning itself over automatically every forty seconds.

. . . . .

△ Besides the keyboard, Version 6 normally supports the use of a mouse. In theory this can have any number of buttons, but since some widely-used computers have single-button mice (e.g., the Apple Macintosh) it's safest not to rely on more than one.

The mouse must be attached to one of the eight windows for it to register properly. (Initially, it isn't attached to any window and so is entirely inert.) To attach it to the window numbered `wnum`, use the opcode:

```
@mouse_window wnum
```

Once attached, a click within the window will register as if it were a key-press to `@read_char` with ZSCII value 254, unless it is a second click in quick succession to a previous click in the same position, in which case it has ZSCII value 253. Thus, a double-clicking registers twice, once as click (254) and then as double-click (253).

At any time, the mouse position, measured in units, and the state of its buttons, i.e., pressed or not pressed, can be read off with the opcode:

```
@read_mouse mouse_array
```

places (x, y) coordinates of the click in `mouse_array-->0` and `mouse_array-->1` and the state of the buttons as a bitmap in `mouse_array-->2`, with bit 0 representing the rightmost button, bit 1 the next and so on. In practice, it's safest simply to test whether this value is zero (no click) or not (click). The array `mouse_array` should have room for 4 entries, however: the fourth relates to menus (see below).

• **EXERCISE 134**
Write a test program to wait for mouse clicks and then print out the state of the mouse.

△ The mouse also allows access to menus, on some interpreters, though in Version 6 only. The model here is of a Mac OS-style desktop, with one or more menus added to the menu bar at the top of the screen: games are free to add or remove menus at any time. They are added with:

```
@make_menu number mtable ?IfAbleTo;
```

Such menus are numbered from 3 upwards. `mtable` is a `table` array of menu entries, each of which must be itself a `string` array giving the text of that option. `IfAbleTo` is a label, to which the interpreter will jump if the menu is successfully created. If `mtable` is zero, the opcode instead removes an already-existing menu. During play, the selection of a menu item by the player is signalled to the Z-machine as a key-press with ZSCII value 252, and the game receiving this can then look up which item on which menu was selected by looking at entry `-->3` in an array given to `read_mouse`. The value in this entry will be the menu number times 256, plus the item number, where items are numbered from 0. If the game has 252 listed as a "terminating character" (see above), then menu selection can take the place of typing a command.

• **EXERCISE 135**
Provide a game with a menu of common commands like "inventory" and "look" to save on typing.

. . . . .

△ The Z-machine can also load and save "auxiliary files" to or from the host machine. These should have names adhering to the "8 + 3" convention, that is, one to eight alphanumeric characters optionally followed by a full stop and one to three further alphanumeric characters. Where no such extension is given, it is assumed to be `.AUX`. Designers are asked to avoid using the extensions `.INF`, `.H`, `.SAV` or `.Z5` or similar, to prevent confusion. Note that auxiliary files from different games may be sharing a common directory on the host machine, so that a filename should be as distinctive as possible. The two opcodes are:

`@save buffer length filename -> R`

Saves the byte array `buffer` (of size `length`) to a file, whose (default) name is given in the `filename` (a `string` array). Afterwards, `R` holds `true` on success, `false` on failure.

`@restore buffer length filename -> R`

Loads in the byte array `buffer` (of size `length`) from a file, whose (default) name is given in the `filename` (a `string` array). Afterwards, `R` holds the number of bytes successfully read.

• **EXERCISE 136**
How might this assist a "role-playing game campaign" with several scenarios, each implemented as a separate Inform game but sharing a player-character who takes objects and experience from one scenario to the next?

• **EXERCISE 137**
Design catacombs in which the ghosts of former, dead players from previous games linger.

. . . . .

△ Finally, the Z-machine supports a very simple form of exception-handling like that of the C language's long jump feature. This is very occasionally useful to get the program out of large recursive tangles in a hurry.

`@catch -> result`

The opposite of `@throw`, `@catch` preserves the "stack frame" of the current routine in the variable `result`: roughly speaking, the stack frame is the current position of which routine is being run and which ones have called it so far.

`@throw value stack-frame`

This causes the program to execute a return with `value`, but as if it were returning from the routine which was running when the `stack-frame` was "caught", that is, set up by a corresponding `@catch` opcode. Note that you can only `@throw` back to a routine which is still running, i.e., to which control will eventually return anyway.

• △△**EXERCISE 138**

Use `@throw` and `@catch` to make an exception handler for actions, so that any action subroutine getting into recursive trouble can throw an exception and escape.

• **REFERENCES**

The assembly-language connoisseur will appreciate 'Freefall' by Andrew Plotkin and 'Robots' by Torbjörn Andersson, although the present lack of on-line hints make these difficult games to win. •Gevan Dutton has made an amazing port of the classic character-graphic maze adventure 'Rogue' to Inform, taking place entirely in the upper window. •Similarly, 'Zugzwang' by Magnus Olsson plots up a chess position. •The function library `"text_functions.h"`, by Patrick Kellum, offers text styling and colouring. These routines are entirely written in assembler. Similar facilities are available from Chris Klimas's `"style.h"` and L. Ross Raszewski's `"utility.h"`. •Jason Penney's `"V6Lib.h"` is a coherent extension to the Inform library for Version 6 games (only), offering support for multiple text windows, images and sounds by means of class definitions and high-level Inform code. •More modestly, but applicably to Version 5 and 8 games, L. Ross Raszewski's `"sound.h"` function library handles sound effects.

§43 Pictures, sounds, blurbs and Blorb

> The blorb spell (safely protect a small object as though in a strong box).
>
> — Marc Blank and P. David Lebling, 'Enchanter'

Pictures may, but need not, accompany a Version 6 game. They are not stored in the story file itself, and different interpreters make different arrangements for getting access to them. Some interpreters can only read low-resolution, low-colour-range images in the awkward format used by Infocom's graphical games. Others take pictures from a ''Blorb file'' which can hold high-resolution and riotously colourful images in a format called PNG. The story file neither knows nor cares which, and refers to pictures only by their numbers.

A Blorb file can also safely protect sound effects and even the story file itself, so that a game and its multi-media resources can be a single file. Blorb is a simple format largely devised by Andrew Plotkin (partly based on the same author's earlier ''Pickle''); it has been fully implemented in Kevin Bracey's 'Zip2000' interpreter for Acorn RISC OS machines, and is also used by the new ''glulx'' format of story files.

A Perl script called perlBlorb, runnable on many models of computer, gathers together sounds and images and constructs Blorb files as needed, from a list of instructions called a ''blurb file''. For instance:

```
! Example of a blurb file
copyright "Angela M. Horns 1998"
release 17
palette 16 bit
resolution 600x400
storyfile "games/sherbet.z5"
sound    creak  "sounds/creaking.snd"
sound    wind   "sounds/wind.snd"
picture  flag     "flag.png"      scale 3/1
picture  pattern  "backdrop.png"
```

When run through perlBlorb, the above produces the text below:

```
! perlBlorb 1.0 [executing on 980124 at 15:31.33]
Constant SOUND_creak = 3;
```

```
Constant SOUND_wind = 4;
Constant PICTURE_flag = 1;
Constant PICTURE_pattern = 2;
! Completed: size 45684 bytes (2 pictures, 2 sounds)
```

This output text looks like Inform source code, and this is not an accident: the idea is that it can be used as an `Include` file to give sensible names to the sound and picture numbers, so that the rest of the code can include statements like this one:

```
@sound_effect SOUND_creak 2 128 255;
```

("start playing this effect at about half maximum volume, repeating it indefinitely"). An attractive alternative is to use a convenient class library, such as `"V6Lib.h"` by Jason Penney, to avoid messing about with assembly language.

You're free to specify the numbering yourself, and you need not give names for the pictures and sounds. A blurb command like:

```
picture "backdrop.png"
```

gives this image the next picture number: i.e., the previous picture number plus 1, or just 1 if it's the first specified picture. On the other hand, a blurb command like:

```
picture 100 "backdrop.png"
```

gives it picture number 100. The only restriction is that pictures must be given in increasing numerical order. The numbering of sounds is similar.

. . . . .

△ The full specification for the "blurb" language is as follows. With one exception (see `palette` below) each command occupies one and only one line of text. Lines are permitted to be empty or to contain only white space. Lines whose first non-white-space character is an exclamation mark are treated as comments, that is, ignored. ("White space" means spaces and tab characters.)

⟨string⟩ means any text within double-quotes, not containing either double-quote or new-line characters

⟨number⟩ means a decimal number in the range 0 to 32767

⟨id⟩ means either nothing at all, or a ⟨number⟩, or a sequence of up to 20 letters, digits or underscore characters _

⟨dim⟩ indicates screen dimensions, and must take the form ⟨number⟩`x`⟨number⟩

⟨ratio⟩ is a fraction in the form ⟨number⟩`/`⟨number⟩. `0/0` is legal but otherwise both numbers must be positive

⟨colour⟩ is a colour expressed as six hexadecimal digits, as in some HTML tags: for instance `F5DEB3` is the colour of wheat, with red value `F5` (on a scale `00`, none, to `FF`, full), green value `DE` and blue value `B3`. Hexadecimal digits may be given in either upper or lower case.

With the exception of `picture` and `sound`, each type of command can only occur at most once in any blurb file. Commands can be used in any order or not at all: an empty "blurb" file results in a perfectly legal, if useless, Blorb file. The full set of commands is as follows:

`copyright` ⟨string⟩

Adds this copyright declaration to the file. It would normally consist of the author's name and the date.

`release` ⟨number⟩

Gives this release number to the file. This is the number returned by the opcode `@picture_data 0` within any game using the Blorb file, and might be used when printing out version information.

`palette 16 bit`
`palette 32 bit`
`palette {` ⟨colour-1⟩ ... ⟨colour-N⟩ `}`

Blorb allows designers to signal to the interpreter that a particular colour-scheme is in use. The first two options simply suggest that the pictures are best displayed using at least 16-bit, or 32-bit, colours. The third option specifies colours used in the pictures in terms of red/green/blue levels, and the braces allow the sequence of colours to continue over many lines. At least one and at most 256 colours may be defined in this way. This is only a "clue" to the interpreter; see the Blorb specification for details.

`resolution` ⟨dim⟩
`resolution` ⟨dim⟩ `min` ⟨dim⟩
`resolution` ⟨dim⟩ `max` ⟨dim⟩
`resolution` ⟨dim⟩ `min` ⟨dim⟩ `max` ⟨dim⟩

Allows the designer to signal a preferred screen size, in real pixels, in case the interpreter should have any choice over this. The minimum and maximum values are the extreme values at which the designer thinks the game will be playable: they're optional, the default values being `0x0` and infinity by infinity.

`storyfile` ⟨string⟩

`storyfile` ⟨string⟩ `include`

Tells perlBlorb the filename of the Z-code story file which these resources are being provided for. (There is no need to do this if you prefer not to.) Usually the Blorb file simply contains a note of the release number, serial code and checksum of the story file, which an interpreter can try to match at run-time to see if the Blorb file and story file go together. If the `include` option is used, however, the entire story file is embedded within the Blorb file, so that game and resources are all bound up in one single file.

`sound` ⟨id⟩ ⟨string⟩
`sound` ⟨id⟩ ⟨string⟩ `repeat` ⟨number⟩
`sound` ⟨id⟩ ⟨string⟩ `repeat forever`
`sound` ⟨id⟩ ⟨string⟩ `music`
`sound` ⟨id⟩ ⟨string⟩ `song`

Tells perlBlorb to take a sound sample from the named file and make it the sound effect with the given number. The file should be an AIFF file unless `music` is specified, in which case it should be a MOD file (roughly speaking a SoundTracker file); or unless `song` is specified, in which case it should be a song file (roughly, a SoundTracker file using other Blorb sound effects as note samples). Note that repeat information (the number of repeats to be played) is meaningful only with version 3 story files using sound effects, as only Infocom's 'The Lurking Horror' ever has.

`picture` ⟨id⟩ ⟨string⟩
`picture` ⟨id⟩ ⟨string⟩ `scale` ⟨ratio⟩
`picture` ⟨id⟩ ⟨string⟩ `scale min` ⟨ratio⟩
`picture` ⟨id⟩ ⟨string⟩ `scale` ⟨ratio⟩ `min` ⟨ratio⟩
and so on

Similarly for pictures: the named file must be a PNG-format image. Optionally, the designer can specify a scale factor at which the interpreter will display the image – or, alternatively, a range of acceptable scale factors, from which the interpreter may choose its own scale factor. (By default an image is not scaleable and an interpreter must display it pixel-for-pixel.) There are three optional scale factors given: the preferred scale factor, the minimum and the maximum allowed. The minimum and maximum each default to the preferred value if not given, and the default preferred scale factor is 1. Scale factors are expressed as fractions: so for instance,

```
picture "flag/png" scale 3/1
```

means ''always display three times its normal size'', whereas

```
picture "backdrop/png" scale min 1/10 max 8/1
```

means ''you can display this anywhere between one tenth normal size and eight times normal size, but if possible it ought to be just its normal size''.

• **REFERENCES**

The Perl script 'perlBlorb' is available from the Inform web-page. •The source code to Kevin Bracey's fully Blorb-compliant standard interpreter 'Zip2000' is public. •Andrew Plotkin has published generic C routines for handling Blorb files. •Numerous utility programs exist which will convert GIF or JPEG images to PNG format, or WAV and MPEG sounds to AIFF.

§44 Case study: a library file for menus

> Yes, all right, I won't do the menu. . . I don't think you realise how long it takes to do the menu, but no, it doesn't matter, I'll hang the picture now. If the menus are late for lunch it doesn't matter, the guests can all come and look at the picture till they are ready, right?
>
> — John Cleese and Connie Booth, *Fawlty Towers*

Sometimes one would like to provide a menu of text options, offered to the player as a list on screen which can be rummaged through with the cursor keys. For instance, the hints display in the "solid gold" edition of Infocom's 'Zork I' shows a list of "Invisiclues": "Above Ground", "The Cellar Area", and so on. Moving a cursor to one of these options and pressing RETURN brings up a sub-menu of questions on the general topic chosen: for instance, "How do I cross the mountains?" Besides hints, many modern games use menu displays for instructions, background information, credits and release notes.

An optional library file called `"Menus.h"` is provided to manage such menus. If you want its facilities then, where you previously included `Verblib`, now write:

```
Include "Verblib";
Include "Menus";
```

And this will make the features of `Menus.h` available. This section describes what these simple features are, and how they work, as an extended example of Z-machine programming.

The designer of this system began by noticing that menus and submenus and options fit together in a tree structure rather like the object tree:

Hints for 'Zork I' (menu)
    ⟶ Above Ground (submenu)
        ⟶ How do I cross the mountains? (option)
            ⟶ *some text is revealed*
    ⟶ The Cellar Area (submenu)
        ⟶ . . .

The library file therefore defines two classes of object, `Menu` and `Option`. The short name of a menu is its title, while its children are the possible choices, which can be of either class. (So you can have as many levels of submenu as needed.) Since choosing an `Option` is supposed to produce some text, which is vaguely like examining objects, the `description` property of an `Option` holds the information revealed. So, for instance:

```
Menu hints_menu "Hints for Zork I";
Menu -> "Above Ground";
Option -> -> "How do I cross the mountains?"
  with description "By ...";
Menu -> "The Cellar Area";
```

Note that such a structure can be rearranged in play just as the rest of the object tree can, which is convenient for "adaptive hints", where the hints offered vary with the player's present travail.

How does this work? A menu or an option is chosen by being sent the message `select`. So the designer will launch the menu, perhaps in response to the player having typed "hints", like so:

```
[ HintsSub;
  hints_menu.select();
];
```

As the player browses through the menu, each menu sends the `select` message to the next one chosen, and so on. This already suggests that menus and options are basically similar, and in fact that's right: `Menu` is actually a subclass of `Option`, which is the more basic idea of the two.

. . . . .

The actual code of `Menus.h` is slightly different from that given below, but only to fuss with dealing with early copies of the rest of the library, and to handle multiple languages. It begins with the class definition of `Option`, as follows:

```
Class Option
 with select [;
          self.emblazon(1, 1, 1);
          @set_window 0; font on; style roman; new_line; new_line;
          if (self provides description) return self.description();
          "[No text written for this option.]^";
      ],
```

The option sends itself the message `emblazon(1,1,1)` to clear the screen and put a bar of height 1 line at the top, containing the title of the option centred. The other two 1s declare that this is "page 1 of 1": see below. Window 0 (the ordinary, lower window) is then selected; text reverts to its usual state of being roman-style and using a variable-pitched font. The screen is now empty and ready for use, and the option expects to have a `description` property which actually does any printing that's required. To get back to the emblazoning:

```
emblazon [ bar_height page pages temp;
    screen_width = 0->33;
    !   Clear screen:
    @erase_window -1;
    @split_window bar_height;
    !   Black out top line in reverse video:
    @set_window 1;
    @set_cursor 1 1;
    style reverse; spaces(screen_width);
    if (standard_interpreter == 0)
        @set_cursor 1 1;
    else {
        ForUseByOptions-->0 = 128;
        @output_stream 3 ForUseByOptions;
        print (name) self;
        if (pages ~= 1) print " [", page, "/", pages, "]";
        @output_stream -3;
        temp = (screen_width - ForUseByOptions-->0)/2;
        @set_cursor 1 temp;
    }
    print (name) self;
    if (pages ~= 1) print " [", page, "/", pages, "]";
    return ForUseByOptions-->0;
];
```

That completes `Option`. However, since this code refers to a variable and an array, we had better write definitions of them:

```
Global screen_width;
Global screen_height;
Array ForUseByOptions -> 129;
```

(The other global variable, `screen_height`, will be used later. The variables are global because they will be needed by all of the menu objects.) The `emblazon` code checks to see if it's running on a standard interpreter. If so, it uses output stream 3 into an array to measure the length of text like "The

Cellars [2/3]" in order to centre it on the top line. If not, the text appears at the top left instead.

So much for `Option`. The definition of `Menu` is, inevitably, longer. It inherits `emblazon` from its superclass `Option`, but overrides the definition of `select` with something more elaborate:

```
Class Menu class Option
 with select [ count j obj pkey line oldline top_line bottom_line
        page pages options top_option;
          screen_width = 0->33;
          screen_height = 0->32;
          if (screen_height == 0 or 255) screen_height = 18;
          screen_height = screen_height - 7;
```

The first task is to work out how much room the screen has to display options. The width and height, in characters, are read out of the story file's header area, where the interpreter has written them. In case the interpreter is *really* poor, we guess at 18 if the height is claimed to be zero or 255; since this is a library file and will be widely used, it errs on the side of extreme caution. Finally, 7 is subtracted because seven of the screen lines are occupied by the panel at the top and white space above and below the choices. The upshot is that `screen_height` is the actual maximum number of options to be offered per page of the menu. Next: how many options are available?

```
          options = 0;
          objectloop (obj in self && obj ofclass Option) options++;
          if (options == 0) return 2;
```

(Note that a `Menu` is also an `Option`.) We can now work out how many pages will be needed.

```
          pages = options/screen_height;
          if (options%screen_height ~= 0) pages++;
          top_line = 6;
          page = 1;
          line = top_line;
```

`top_line` is the highest screen line used to display an option: line 6. The local variables `page` and `line` show which line on which page the current selection arrow points to, so we're starting at the top line of page 1.

```
          .ReDisplay;
          top_option = (page - 1) * screen_height;
```

This is the option number currently selected, counting from zero. We display the three-line black strip at the top of the screen, using `emblazon` to create the upper window:

```
self.emblazon(7 + count, page, pages);
@set_cursor 2 1; spaces(screen_width);
@set_cursor 2 2; print "N = next subject";
j = screen_width-12; @set_cursor 2 j; print "P = previous";
@set_cursor 3 1; spaces(screen_width);
@set_cursor 3 2; print "RETURN = read subject";
j = screen_width-17; @set_cursor 3 j;
```

The last part of the black strip to print is the one offering Q to quit:

```
if (sender ofclass Option) print "Q = previous menu";
else print "  Q = resume game";
style roman;
```

The point of this is that pressing Q only takes us back to the previous menu if we're inside the hierarchy, i.e., if the message `select` was sent to this `Menu` by another `Option`; whereas if not, Q takes us out of the menu altogether. Next, we count through those options appearing on the current page and print their names.

```
count = top_line; j = 0;
objectloop (obj in self && obj ofclass Option) {
    if (j >= top_option && j < (top_option+screen_height)) {
        @set_cursor count 6;
        print (name) obj;
        count++;
    }
    j++;
}
bottom_line = count - 1;
```

Note that the name of the option begins on column 6 of each line. The player's current selection is shown with a cursor > appearing in column 4:

```
oldline = 0;
for (::) {
    ! Move or create the > cursor:
    if (line ~= oldline) {
        if (oldline ~= 0) {
            @set_cursor oldline 4; print " ";
        }
        @set_cursor line 4; print ">";
```

```
        }
        oldline = line;
```

Now we wait for a single key-press from the player:

```
        @read_char 1 -> pkey;
        if (pkey == 'N' or 'n' or 130) {
            ! Cursor down:
            line++;
            if (line > bottom_line) {
                line = top_line;
                if (pages > 1) {
                    if (page == pages) page = 1; else page++;
                    jump ReDisplay;
                }
            }
            continue;
        }
```

130 is the ZSCII code for "cursor down key". Note that if the player tries to move the cursor off the bottom of the list, and there's at least one more page, we jump right out of the loop and back to `ReDisplay` to start again from the top of the next page. Handling the "previous" option is very similar, and then:

```
        if (pkey == 'Q' or 'q' or 27 or 131) break;
```

Thus pressing lower or upper case Q, escape (ZSCII 27) or cursor left (ZSCII 131) all have the same effect: to break out of the `for` loop. Otherwise, one can press RETURN or cursor right to select an option:

```
        if (pkey == 10 or 13 or 132) {
            count = 0;
            objectloop (obj in self && obj ofclass Option) {
                if (count == top_option + line - top_line) break;
                count++;
            }
            switch (obj.select()) {
                2: jump ReDisplay;
                3: jump ExitMenu;
            }
            print "[Please press SPACE to continue.]^";
            @read_char 1 -> pkey;
            jump ReDisplay;
        }
    }
```

(No modern interpreter should ever give 10 for the key-code of RETURN, which is ZSCII 13. Once again, the library file is erring on the side of extreme caution.) An option's `select` routine can return three different values for different effects:

2	Redisplay the menu page that selected me
3	Exit from that menu page
anything else	Wait for SPACE, then redisplay that menu page

Finally, the exit from the menu, either because the player typed Q, escape, etc., or because the selected option returned 3:

```
            .ExitMenu;
            if (sender ofclass Option) return 2;
            font on; @set_cursor 1 1;
            @erase_window -1; @set_window 0;
            new_line; new_line; new_line;
            if (deadflag == 0) <<Look>>;
            return 2;
        ];
```

And that's it. If this menu was the highest-level one, it needs to resume the game politely, by clearing the screen and performing a `Look` action. If not, then it needs only to return 2, indicating "redisplay the menu page that selected me": that is, the menu one level above.

The only remaining code in `"Menus.h"` shows some of the flexibility of the above design, by defining a special type of option:

```
Class SwitchOption class Option
  with short_name [;
            print (object) self, " ";
            if (self has on) print "(on)"; else print "(off)";
            rtrue;
       ],
       select [;
            if (self has on) give self ~on; else give self on;
            return 2;
       ];
```

Here is an example of `SwitchOptions` in use:

```
Menu settings "Game settings";
SwitchOption -> FullRoomD   "full room descriptions" has on;
SwitchOption -> WordyP      "wordier prompts";
SwitchOption -> AllowSavedG "allow saved games" has on;
```

So each option has the attribute `on` only if currently set. In the menu, the option `FullRoomD` is displayed either as "full room descriptions (on)" or "full room descriptions (off)", and selecting it switches the state, like a light switch. The rest of the code can then perform tests like so:

```
if (AllowSavedG hasnt on) "That spell is forbidden.";
```

. . . . .

Appearance of the final menu on a screen 64 characters wide:

```
line 1                          Hints for Zork I [1/2]
line 2   N = next subject                                P = previous
line 3   RETURN = read subject                         Q = resume game
line 4
line 5
line 6       Above Ground
line 7     > The Cellar Area
line 8       The Maze
line 9       The Round Room Area
```

• **REFERENCES**

Because there was a crying need for good menus in the early days of Inform, there are now numerous library extensions to support menus and interfaces built from them. The original such was L. Ross Raszewski's `"domenu.h"`, which provides a core of basic routines. `"AltMenu.h"` then uses these routines to emulate the same menu structures coded up in this section. `"Hints.h"` employs them for Invisiclues-style hints; `"manual.h"` for browsing books and manuals; `"converse.h"` for menu-based conversations with people, similar to those in graphical adventure games. Or indeed to those in Adam Cadre's game 'Photopia', and Adam has kindly extracted his menu-based conversational routines into an example program called `"phototalk.inf"`. For branching menus, such as a tree of questions and answers, try Chris Klimas's `"branch.h"`. To put a menu of commands at the status line of a typical game, try Adam Stark's `"action.h"`.

§45 Limitations and getting around them

> How wide the limits stand
> Between a splendid and an happy land.
> — Oliver Goldsmith (1728–1774), *The Deserted Village*

The Z-machine is well-designed, and has three major advantages: it is compact, widely portable and can be quickly executed. Nevertheless, like any rigidly defined format it imposes limitations. This section is intended to help those few designers who encounter the current limits. Some of the economy-measures below may sound like increasingly desperate manoeuvres in a lost battle, but if so then the cavalry is on its way: Andrew Plotkin has written a hybrid version of Inform which removes almost every restriction. Although it doesn't quite have all the nooks and crannies of Inform yet working, it does allow most games to compile without difficulty to a very much larger virtual machine than the Z-machine called "glulx".

1. Story file size. The maximum size of a story file (in K) is given by:

V3	V4	**V5**	**V6**	V7	**V8**
128	256	256	512	320	512

Because the centralised library of Inform is efficient in terms of not duplicating code, even 128K allows for a game at least half as large again as a typical old-style Infocom game. Inform is normally used only to produce story files of Versions 5, 6 and 8. Version 5 is the default; Version 6 should be used where pictures or other graphical features are essential to the game; Version 8 is a size extension for Version 5, allowing games of fairly gargantuan proportions.

△ If story file memory does become short, a standard mechanism can save about 8–10% of the total memory, though it will not greatly affect readable memory extent. Inform does not usually trouble with this economy measure, since there's very seldom any need, and it makes the compiler run about 10% slower. What you need to do is define abbreviations and then run the compiler in its "economy" mode (using the switch `-e`). For instance, the directive

```
Abbreviate " the ";
```

(placed before any text appears) will cause the string " the " to be internally stored as a single 'letter', saving memory every time it occurs (about 2,500 times in 'Curses', for

instance). You can have up to 64 abbreviations. When choosing abbreviations, avoid proper nouns and instead pick on short combinations of a space and common two- or three-letter blocks. Good choices include `" the "`, `"The "`, `", "`, `" and "`, `"you"`, `" a "`, `"ing "`, `" to"`. You can even get Inform to work out by itself what a good stock of abbreviations would be, by setting the `-u` switch: but be warned, this makes the compiler run about 29,000% slower.

2. Readable memory size. In a very large game, or even a small one if it uses unusually large or very many arrays, the designer may run up against the following Inform fatal error message:

> This program has overflowed the maximum readable-memory size of the Z-machine format. See the memory map below: the start of the area marked "above readable memory" must be brought down to `$10000` or less.

In other words, the readable-memory area is absolutely limited to 65,536 bytes in all Versions. Using the `-D` debugging option increases the amount of readable-memory consumed, and the Infix `-X` switch increases this further yet. (For instance 'Advent' normally uses 24,820 bytes, but 25,276 with `-D` and 28,908 with `-X`.) The following table suggests what is, and what is not, worth economising on.

Each...	*Costs...*
Routine	0
Text in double-quotes	0
Object or class	26
Common property value	3
Non-common property value	5
If a property holds an array	add 2 for each entry after the first
Dictionary word	9
Verb	3
Different action	4
Grammar token	3
`-->` or `table` array entry	2
`->` or `string` array entry	1

To draw some morals: verbs, actions, grammar and the dictionary consume little readable memory and are too useful to economise on. Objects and arrays are where savings can be made. Here is one strategy for doing so.

2a. Economise on arrays. Many programmers create arrays with more entries than needed, saying in effect "I'm not sure how many this will take, but it's bound to be less than 1,000, so I'll say 1,000 entries to be on the safe

side." More thought will often reduce the number. If not, look at the typical contents. Are the possible values always between 0 and 255? If so, make it a `->` or `string` array and the consumption of readable memory is halved. Are the possible values always `true` or `false`? If so, Adam Cadre's `"flags.h"` library extension offers a slower-access form of array but which consumes only about 1/8th of a byte of readable memory per array entry.

2b. Turn arrays of constants into routines. Routines cost nothing in readable memory terms, but they can still store information as long as it doesn't need to vary during play. For instance, 'Curses' contains an array beginning:

```
Array RadioSongs table
  "Queen's ~I Want To Break Free~."
  "Bach's ~Air on a G-string~."
  "Mozart's ~Musical Joke~."
```

and so on for dozens more easy-listening songs which sometimes play on Aunt Jemima's radio. It might equally be a routine:

```
[ RadioSongs n;
  switch (n) {
      0: return 100; ! Number of songs
      1: return "Queen's ~I Want To Break Free~.";
      2: return "Bach's ~Air on a G-string~.";
      3: return "Mozart's ~Musical Joke~.";
```

and so on. Instead of reading `RadioSongs-->x`, one now reads `RadioSongs(x)`. Not an elegant trick, but it saves 200 bytes of readable memory.

2c. Economise on object properties. Each time an object provides a property, readable memory is used. This is sometimes worth bearing in mind when writing definitions of classes which will have many members. For instance:

```
Class Doubloon(100)
 with name 'gold' 'golden' 'spanish' 'doubloon' 'coin' 'money'
           'coins//p' 'doubloons//p',
      ...
```

Each of the hundred doubloons has a `name` array with eight entries, so 1700 bytes of readable memory are consumed. This could be reduced to 300 like so:

```
Class Doubloon(100)
 with parse_name [;
      ! A routine detecting the same name-words
      ...
      ],
```

2d. Make commonly occurring properties common. Recall that properties declared with the `Property` directive are called "common properties": these are faster to access and consume less memory. If, for instance, each of 100 rooms in your game provides a property called `time_zone`, then the declaration

```
Property time_zone;
```

at the start of your code will save 2 bytes each time `time_zone` is provided, saving 200 bytes in all. (The library's properties are all common already.)

2e. Economise on objects. In a room with four scenery objects irrelevant to the action, say a window, a chest of drawers, a bed and a carpet, is it strictly necessary for each to have its own object? Kathleen Fischer: "`parse_name` is your friend... a single object with an elaborate `parse_name` can be used to cover a whole lot of scenery." In Kathleen's technique, it would use `parse_name` to record which of the words "window", "chest", "bed" or "carpet" was used, storing that information in a property: other properties, like `description`, would be routines which produced text depending on what the object is representing this turn.

2f. Reuse objects. This is a last resort but L. Ross Raszewski's `"imem.h"` has helped several designers through it. Briefly, just as an array was converted to a routine in (1) above, `"imem.h"` converts object definitions to routines, with a minimal number of them "swapped in" as real objects at any given time and the rest – items of scenery in distant locations, for instance – "swapped out".

3. Grammar. There can be up to 256 essentially different verbs, each with up to 32 grammar lines. Using the `UnknownVerb` entry point will get around the former limit, and general parsing routines can make even a single grammar line match almost any range of syntax.

4. Vocabulary. There is no theoretical limit except that the dictionary words each take up 9 bytes of readable memory, which means that 4,000 words is probably the practical limit. In practice games generally have vocabularies of between 500 and 2,000 words.

5. Dictionary resolution. Dictionary words are truncated to their first 9 letters (except that non-alphabetic characters, such as hyphens, count as 2 "letters" for this purpose: look up `Zcharacter` in the index for references to more on this). Upper and lower case letters are considered equal. Since general parsing routines, or `parse_name` routines, can look at the exact text typed by the player, finer resolution is easy enough if needed.

6. Attributes, properties, names. There can be up to 48 attributes and an unlimited number of properties, at most 63 of these can be made common by

being declared with `Property`. A property entry can hold up to 64 bytes of data. Hence, for example, an object can have up to 32 `name`s. If an object must respond to more, give it a suitable `parse_name` routine.

7. Objects and classes. The number of objects is unlimited so long as there is readable memory to hold their definitions. The number of classes is presently limited to 256, of which the library uses only 1.

8. Global variables. There can only be 240 of these, and the Inform compiler uses 5 as scratch space, while the library uses slightly over 100; but since a typical game uses only a dozen of its own, code being almost always object-oriented, the restriction is never felt.

9. Arrays. An unlimited number of `Array` statements is permitted, although the entries in arrays consume readable memory (see above).

10. Function calls and messages. A function can be called with at most seven arguments. A message can be called with at most five.

11. Recursion and stack usage. The limit on this is rather technical (see *The Z-Machine Standards Document*). Roughly speaking, recursion is permitted to a depth of 90 routines in almost all circumstances, and often much deeper. Direct usage of the stack via assembly language must be modest.