Inform Release Notes
–
25 November 2003 (Beta)

Inform Compiler 6.30
Inform Library 6/11

Introduction

This is a maintenance release of the Inform system for creating adventure games, intended to address issues that have arisen since the 6.21 compiler and 6/10 library files were released in 1999. The focus is primarily on fixing the problems which are identified in the Inform Patch List:

```
http://www.inform-fiction.org/patches/index.html
```

In addition, a small number of enhancements are included, based on the ideas collected in the Inform Suggestions List:

```
http://www.firthworks.com/roger/suggest.html
```

Although just about all known bugs are fixed, the approach to enhancing Inform is more conservative. The selection of suggestions to implement has been governed by three factors:

- avoidance of changes which might cause existing games to misbehave;
- minimisation of features which would require updates to the *Inform Designer's Manual*;
- essential simplicity. With Graham Nelson's permission, this release has been produced by a volunteer taskforce, whose enthusiasm has been tempered by a certain lack of familiarity with the internals of Inform. Thus, we have concentrated on 'safe' changes; the implementation of some good ideas has been postponed until we are more confident that we know what we're doing.

Having said that, this release does incorporate one major advance. It is based on Andrew Plotkin's bi-platform – Z-machine and Glulx – compiler and library files, which were in turn derived from the the 6.21 compiler and 6/10 library. The result is that the two Virtual Machine (VM) strands have merged into a single compiler and library which, although continuing by default to produce Z-code, can alternatively generate code for the Glulx VM if you supply the `-G` compiler switch. There's more on this topic in "Support for Glulx" on page 13. Before that, though, we'll summarise the changes to the compiler and to the library files.

Acknowledgements

Far too many people contributed towards this release – reporting and resolving bugs, making helpful suggestions, providing support and facilities, testing, and so on – for their names to be individually listed. So instead, this is a general thank-you to everybody who has made this release happen, and specific ones to Graham for permitting it in the first place, and to Andrew for his pioneering work on Glulx.

There are, naturally, sure to be errors in the release. Please continue to report small problems through the Patch List in the usual way; more serious issues should perhaps be raised on the rec.arts.int-fiction Usenet newsgroup, with a Subject line beginning "`[Inform63] ...`".

Compiler 6.30

These are the changes delivered in version 6.30 of the Inform compiler. See:

Features added

Several new features are available.

- The compiler automatically defines a `WORDSIZE` constant, whose value is 2 when compiling for the Z-machine, and 4 when the target is Glulx. The constant specifies the number of bytes in a VM word, and we recommend that you use it in the small number of circumstances where this value is significant. The compiler also defines a constant `TARGET_GLULX` if you supply the **`-G`** switch, or `TARGET_ZCODE` otherwise; in both cases the constant value is 0. For more information on the use of these constants, see "Support for Glulx" on page 13.
- The `Switches` directive, which enables certain compiler switches to be set from within the source file rather than on the compiler command line, has been superseded by a more powerful mechanism. The special comment characters "`!%`", occurring on the very first line or lines of the source file, enable you to specify Inform Command Language (ICL) commands to control the compilation. For example:

```
!% -E1G                                        ! Glulx, 'Microsoft' errors
!% -~S                                         ! disable Strict mode
!% +include_path=./test,../,../lib/contrib     ! look in 'test' library
!% $MAX_STATIC_MEMORY=20000
Constant STORY "RUINS";
...
```

  ICL is described in §39 of the *Inform Designer's Manual*. In brief: each line specifies a single command, starting with "`-`" to define one or more switches, "`+`" to define a path variable, or "`$`" to define a memory setting. Comments are introduced by "`!`". The ICL command "`compile`" is not permitted at the head of a source file.

- An new directive, similar to `Array ... string` and `Array ... table`, is provided:

```
Array array buffer N;
Array array buffer expr1 expr2 ... exprN;
Array array buffer "string";
```

  This creates a hybrid array of the form used by *`string`*`.print_to_array()` and the new library routine `PrintToBuffer()`, in which the first **word** *`array`*`-->0` contains *`N`* and the following *`N`* **bytes** *`array`*`->WORDSIZE`, *`array`*`->(WORDSIZE+1)` ... *`array`*`->(WORDSIZE+`*`N`*`-1)` contain the specified expression values or string characters.

- A new `(A)` print rule – similar to the existing `(The)` – prints an object's indirect article with its first letter capitalised. The printed article is "A" or "An" by default, or else taken from the object's `article` property.
- The minimum size of the Z-code header extension table can be set using the command line switch **`-W`***`n`*. For example, **`-W6`** makes the table at least six words long.

- Source code in character sets other than ISO 8859-1 to 8859-9 is now supported, provided that the character set can be mapped onto one of the ISO 8859 sets.

  A mapping file is used to define how the source code is to be processed. This file consists of a directive indicating the ISO 8859 set to be mapped to, followed by 256 numbers giving the mapping. As an example, under Microsoft Windows, Russian text is encoded with the character set defined as Microsoft code page 1251. The following text defines a mapping to the ISO 8859-5 set:

```
! Windows Cyrillic (code page 1251) to ISO 8859-5
C5
  0, 63, 63, 63, 63, 63, 63, 63, 63, 32, 10, 63, 10, 10, 63, 63
 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63
 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63
 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79
 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95
 96, 97, 98, 99,100,101,102,103,104,105,106,107,108,109,110,111
112,113,114,115,116,117,118,119,120,121,122,123,124,125,126, 63
162,163, 44,243, 34, 46, 63, 63, 63, 63,169, 60,170,172,171,175
242, 39, 39, 34, 34, 46, 45, 45,152, 84,249, 62,250,252,251,255
 32,174,254,168, 36, 63,124,253,161, 67,164, 60, 63, 45, 82,167
 63, 63,166,246, 63, 63, 63, 46,241,240,244, 62,248,165,245,247
176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191
192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207
208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223
224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239
```

  Lines starting with "!" are treated as comments. The next line, beginning with "C", defines the ISO set to map to in the same way as the **-C***n* command line switch.

  To use the mapping, Inform treats each character in the source file as a number between 0 and 255, and uses that number as an index into the mapping table. For example, suppose that the character read in from a Russian Windows text file is the Cyrillic small letter "ya".This character is represented in the Russian Windows character set by the number 255. Inform takes that entry in the mapping, which is 239. Therefore the character is regarded as being 239 in ISO 8859-5.

  The name of the mapping file is specified by a new compiler path variable `+charset_map`. If the above mapping is held in a text file `win1251.map`, a Russian game could be compiled with a command line of the form:

```
inform +charset_map=win1251.map +language_name=Russian mygame.inf
```

- The `@check_unicode` and `@print_unicode` opcodes, introduced in the *Z-Machine Standards Document* version 1.0, can now be called by name rather than by using the clumsier generic syntax `@"EXT:11S"` and `@"EXT:12S"`. For example:

```
@print_unicode $0401;
```

- Strict mode (which compiles run-time integrity checks into your game) has been decoupled from Debug mode (which defines debugging verbs like TRACE and SHOWOBJ). This means that it's no longer necessary to turn off Strict checking (in order to disable the Debug verbs) before releasing a game, though of course you can do so if you wish to save space and increase performance. By default, Strict mode is **enabled** (turn it off with **-~S**) and Debug mode is **disabled** (turn it on with **-D**).

- The compiler now issues a warning if you use `array->n` on an array of words, or `array-->n` on an array of bytes. Use the new `Array ... buffer` directive to create a hybrid array which can be accessed without these warnings.
- The compiler also issues a warning if you refer to an unqualified property name within a routine, for example by typing `number=0;` when you intended `self.number=0;`.

Bugs fixed

Items of the form [C62126] and [G03701] quote the bug's reference number in the 'Compiler' section of the Inform Patch List.

- After using `Extend only` to separate off an element of an existing Verb definition, new synonyms for the separated verb now work correctly; previously they were applied to the residue of the original definition. [C62126]
- Strict mode now tests for the use of `@put_prop` or `@get_prop` opcodes when a common property is longer than two bytes – the *Z-Machine Standards Document* says that this is illegal, and that the result is unspecified. [C62125]
- Handling of European quotes is (finally) correct: the "«" symbol is produced by any of `@<<`, `@@163` and `@{00AB}`, while any of `@>>`, `@@162` and `@{00BB}` produce the matching "»". Note, however, that this problem originated in an error in the previous version of the *Z-Machine Standards Document*, and therefore older interpreters written to that specification, or more recent ones adjusted to work with the incorrect fix introduced at Inform 6.12, may still not give the correct results. [C62124]
- The "no such constant" compilation error message now quotes the number of the appropriate source line. [C62123]
- The `metaclass()` and `ZRoutine()` routines no longer report large unsigned values – above the game's Static memory area – as of type String. More usefully, the constant `NULL` (-1) is not reported as a String. [C62122]
- Complex expressions combining a routine call and the `ofclass` and `or` operators no longer generate incorrect code. [C62121]
- Negative constants in assembly operations – for example, `@set_colour 4 (-1);` – no longer cause the compiler to report an unexpected expression. [C62119]
- Various problems with handling ISO 8859 characters in the range 128-255, and also in the use of @@ escape sequences, have been resolved. [C62117, C62115, C6211, C62003]
- An `Abbreviate` directive containing a substring of "<unknown attribute>" may crash the compiler; hopefully, no more. [C62116]
- The 320Kb size limit placed by Inform on v6 and v7 games has been raised to 512Kb. [C62114]
- Following a `Zcharacter` directive replacing the entire alphabet table, dictionary entries are no longer corrupted. [C62113]
- The compiler now generates conditional branches spanning up to 63 bytes, lifting the previous unnecessary limit of a 31-byte span and leading to slightly shorter code. [C62112]
- Putting an object in itself doesn't now loop indefinitely. [C62110]
- Various problems with the `@store`, `@inc_chk`, `@dec_chk`, `@not` and `@je` opcodes have been resolved. [C62109, C62108, C62105]

- Problems with nested conditional compilation directives `#Ifndef...#Ifnot...#Endif` have been resolved. [C62107]
- A long dictionary word – such as `'elephants//p'` – now correctly sets the plural bit. [C62103]
- When compiling for Glulx, the compiler uses the `@callf`, `@callfi`, `@callfii` or `@callfiii` opcodes where applicable for generated calls instead of always pushing arguments onto the stack and using `@call`. [G03701]
- The presence of a `Switches G;` directive no longer causes the compiler to crash.

Library 6/11

These are the changes delivered in version 6/11 of the Inform library. See:

- "Features added" below
- "Library routines" on page 9
- "Bugs fixed" on page 11

Features added

Several new features are available.

- The library automatically defines four constants: `LIBRARY_PARSER` at the end of Parser.h, `LIBRARY_VERBLIB` at the end of VerbLib.h, `LIBRARY_GRAMMAR` at the end of Grammar.h, and `LIBRARY_ENGLISH` at the end of English.h. Contributed library extensions can use these constant to check that they have been `Included` in the correct location.
- The word "wall" has been removed from the `CompassDirection` objects defined in English.h, whose names are now simply "north", "south", etc.
- The `selfobj` object now includes an empty `add_to_scope` property, which you can over-ride with your own routine, typically to equip the player with body parts. For a single object:

  ```
  selfobj.add_to_scope = nose;
  ```

  or for multiple objects:

  ```
  [ IncludeBodyParts; PlaceInScope(nose); PlaceInScope(hands); ];

  selfobj.add_to_scope = IncludeBodyParts;
  ```
- The verbs RECORDING [ON|OFF] and REPLAY are now always available, irrespective of the DEBUG state. This may cause compilation errors if you have already defined these verbs yourself.
- The verbs PRY, PRISE, PRIZE and LEVER have been added. This may cause compilation errors if you have already defined these verbs yourself.
- The parser treats input lines beginning with "*" as a comment, without attempting any further parsing. The character used to introduce comments can be changed by defining `COMMENT_CHARACTER` before you `Include Parser`. For example:

  ```
  Constant COMMENT_CHARACTER '!';
  ```
- The task-based scoring system (§22 of the *Inform Designer's Manual*) uses a byte array, which precludes the awarding of large or negative scores. To get round this, you can `Replace` the `TaskScore()` library routine as follows, and then define `task_scores` as a **word** array:

  ```
  Replace TaskScore;

  Array task_scores --> 100 200 300 400 (-50) 600;

  [ TaskScore i; return task_scores-->i; ];
  ```
- The scoring system is completely disabled if you define a constant `NO_SCORE` near the start of your game.

  ```
  Constant NO_SCORE;
  ```

- An object's `invent` property – if it has one – is now invoked both when displaying the player's inventory **and** when including the object in a room description. `invent` is invoked in the usual way (with `inventory_stage` first set to 1, and then set to 2) both when mentioning the object in a room description, and when listing it in the player's inventory. By default you'll get the same output each time. If you need to distinguish between the two occasions, you can test `(c_style&PARTINV_BIT)` – true during a room description – or `(c_style&FULLINV_BIT)` – true during an inventory. Here's an example:

```
Object  -> "sack"
  with  name 'sack',
        invent [;
            ! When listing objects in the player's inventory
            if (c_style&FULLINV_BIT) rfalse;

            ! When listing objects at the end of a room description
            if (inventory_stage == 1) switch (children(self)) {
                0: print "an empty sack";
                1: print "a sack containing ", (a) child(self);
                default: print "an assortment of objects in a sack";
                }
            rtrue;
            ],
  has   container open;
```

  This enhancement uses the mechanism described at:

```
http://www.firthworks.com/roger/informfaq/ww.html#4
```

  and means that you no longer need to `Include WriteList`.

- A new `before_implicit` property is available; at the moment this is used only by the parser, when it is about to perform an implicit TAKE (for example, EAT APPLE when you're not holding the apple). You can give this property to an object if you wish to control the parser's behaviour. The property's value should be a routine which returns: 0 to report "(first taking the ...)" and then attempt to do so (this is what currently happens); 1 to attempt the TAKE without first issuing the message, or 2 to proceed with the requested action without attempting the TAKE. The object can test `action_to_be` to determine which action has triggered the TAKE:

```
before_implicit [;
  Take: if (action_to_be == ##Eat) return 2;
],
```

- The `turns` counter is now initialised to 0, not 1. You can change this if you define a constant `START_MOVE` near the start of your game.

```
Constant START_MOVE 1;
```

- A new system variable `sys_statusline_flag` is set to 1 initially if you have used the `statusline time;` directive in your program to show a clock, and to 0 otherwise. It can be changed by the program.

Library routines

Several new library routines are provided to harmonise commonly-encountered differences between the Z-machine and Glulx VMs (see also “Library routines available only in Glulx” on page 18).

`KeyCharPrimitive()`

waits for a single key, and returns the character from 0-255 (or, for Glulx, one of the Glk special key codes.). For Glulx only, an extended form is available – see “Library routines available only in Glulx” on page 18.

`ClearScreen()`

clears both the status line and the main window. The cursor moves to the top of the screen. The routine should be followed by a call to `MoveCursor()` or `MainWindow()`.

`MoveCursor()`
`MoveCursor(`*`line`*`, `*`column`*`)`

`MoveCursor()` selects the status line for output. `MoveCursor(`*`line`*`, `*`column`*`)` selects the status line for output and moves the cursor to the given *`line`* and *`column`* within the status area, where line 1 is the top line and column 1 is the far left. (This is necessary because the Glk convention is to number both lines and columns from 0 rather than 1.)

`MainWindow()`

selects the main (buffered) text window for output.

`StatusLineHeight(`*`lines`*`)`

sets the height of the statusline in lines. The standard `DrawStatusLine()` calls this every turn, which isn’t a bad thing, since `StatusLineHeight()` is smart. If you replace `DrawStatusLine()`, maintain this convention. (The library menu routines fiddle with the status line, and it’s up to `DrawStatusLine()` to reset it after the menus are over.)

A new library variable `gg_statuswin_cursize` holds the current setting (in both VMs).

`ScreenWidth()`

returns the number of characters that can be printed in a monospaced font between the left and right borders of the currently selected window. For Glulx only, the extended form `ScreenWidth(`*`win`*`)` works on a specified window id; note that the results are unreliable if the normal style for that window uses a proportional font.

`ScreenHeight()`

returns the height in lines of the main window.

`SetColour(`*`fg`*`, `*`bg`*`)`
`SetColour(`*`fg`*`, `*`bg`*`, `*`selector`*`)`

`SetColour(`*`fg`*`, `*`bg`*`)` sets the current foreground and background text colours, using the same codes as the `@set_colour` opcode in the Z-machine (1=default, 2=black, 3=red, 4=green, 6=blue etc.). Using `SetColour(`*`fg`*`, `*`bg`*`, `*`selector`*`)`, colours can be set separately in each window: if *`selector`* is 0, both are set (the top window will have inverted colours for the Z-machine); if *`selector`* is 1 only the statusline is affected; if *`selector`* is 2 only the main window is affected.

All colour functions are effective only if the library variable `clr_on` is set to non-zero.

The advantage over `@set_colour` is that when the player restores a saved game or types UNDO, the colours will be correct for that state of the game.

For Glulx, the routine produces an appropriate effect if style hints are enabled by the interpreter; it also clears the screen. For the Z-machine, a separate call to `ClearScreen()` is required.

`DecimalNumber(`*`num`*`)`

prints *num* as a decimal number (it is in fact identical to `print` *num*;). It may be useful in conjunction with...

`PrintToBuffer(`*`array`*`, `*`arraylen`*`, `*`string`*`)`
`PrintToBuffer(`*`array`*`, `*`arraylen`*`, `*`obj`*`)`
`PrintToBuffer(`*`array`*`, `*`arraylen`*`, `*`obj`*`, `*`prop`*`)`
`PrintToBuffer(`*`array`*`, `*`arraylen`*`, `*`routine`*`, `*`arg1`*`, `*`arg2`*`)`

prints its arguments – a string, an object's name, the value of an object's property, or a routine with up to two arguments – to the buffer *array*. The number of characters written to the buffer is a word at *array*`-->0`; the actual characters start at *array*`->WORDSIZE`. The maximum number of characters is specified in *arraylen*; for the Z-machine, an overrun caused by printing more than this value will produce an error message that you have corrupted the contents of memory beyond the array (for Glulx, the output is automatically truncated at the specified *arraylen*).

For Glulx, see also `PrintAnyToArray()` in "Library routines available only in Glulx" on page 18, which has slightly extended capabilities, and which returns the number of characters written rather than writing them at *array*`-->0`.

`Length(`*`string`*`)`

returns the number of characters in the string. Note that this prints to one of the parser arrays, and therefore it is your responsibility to ensure that the length **cannot be greater that 160 characters**.

`UpperCase(`*`char`*`)`
`LowerCase(`*`char`*`)`

return *char* in upper or lower case (if it was alphabetic), or unchanged (otherwise). Changes affecting A-Z and a-z are always reliable; changes to accented characters will not work if you have supplied a compiler switch **`-C2`** through **`-C9`**, or used a `Zcharacter` directive to adjust the standard ZSCII character set.

`PrintCapitalised(`*`obj`*`, `*`prop`*`, `*`flag`*`, `*`nocaps`*`)`

is based upon `PrintOrRun(`*`obj`*`, `*`prop`*`, `*`flag`*`)`. `PrintOrRun()` tests *obj.prop*, and either runs it (if a Routine), or prints it (if a String). In the latter case, a newline is then output unless *flag* is `true`. `PrintCapitalised()` does all that; the difference is that the first letter of any output is in upper case unless *nocaps* is `true`.

`Cap(`*`string`*`, `*`nocaps`*`)`

prints the *string* with the first letter in upper case, unless *nocaps* is true. Can also be used as a print rule:

```
print ..., (Cap) myString, ...;
```

`Centre(`*`string`*`)`

prints a single-line *`string`* approximately centrally between the left and right borders of the screen by preceding it with an appropriate number of spaces. The routine works only for monospaced fonts (that is, after `font off;`), and is only likely to work well in the main Glulx window if the normal style for TextBuffer uses a non-proportional font. It is however useful for centring information in the status line. Can also be used as a print rule:

```
print ..., (Centre) myString, ...;
```

Bugs fixed

Items of the form [L61034] quote the bug's reference number in the 'Library' section of the Inform Patch List.

- A command like EMPTY ME no longer replies "yourself can't contain things". [L61036]
- The commands TAKE ALL FROM X and REMOVE ALL FROM X, where X is a closed or empty container, now produce "There are none at all available" which, while maybe not ideal, is better than "You can't see any such thing" and "You can't use multiple objects with that verb" respectively. [L61035]
- A problem with the misbehaviour of `name` properties on rooms, in conjunction with THE, has been corrected. [L61034]
- The command PUT X INTO X now correctly produces "You can't put something inside itself", rather than "You can't see any such thing". [L61033]
- Run-time errors resulting from `IndirectlyContains()` attempting to find the parent of a Class which supports dynamic creation of objects have been resolved. [L61032]
- Code in `Parser__parse()` which deals with looking ahead to the indirect object in cases like PUT ALL INTO BAG (a MULTIEXCEPT token) and TAKE ALL FROM BAG (a MULTIINSIDE token) now correctly sets the `advance_warning` global (to BAG). [L61031, L61023]
- The *Inform Designer's Manual* (p. 98) states that SHOWOBJ should accept an object number; now it does. [L61030]
- The `YesOrNo()` routine now re-prompts correctly after garbage input. [L61029]
- The parse buffer is no longer declared and initialised incorrectly (albeit harmlessly). [L61028, L60708]
- The *Inform Designer's Manual* (p. 93) defines the calling order of routines and properties for the 'Before' stage as follows:

  1. `GamePreRoutine()`
  2. `orders` of the player
  3. `react_before` of every object in scope
  4. `before` of the current room
  5. `before` of the first noun, if specified

  In the library, however, steps 3 and 4 are executed in reverse order. They are now as documented. [L61027]
- A `found_in` floating object which the player is able to take (probably due to a coding error) is no longer silently dropped when the player returns to one of the listed rooms. [L61026]

- A small problem with inherited `describe` properties has been corrected. [L61025]
- Standard screen-handling is now implemented in v6 games. [L61022]
- The handling of “You can’t go that way” messages is made consistent. Also, the statement `ChangeDefault(cant_go,myRoutine);` now works. [L61020]
- Attempting to place an object in/on an object where it is already now results in “It’s already there”, rather than “You need to be holding it before you can put it into something else”. [L61019]
- A problem with misleading inventory listing has be clarified. [L61018]
- The command LEAVE X now correctly produces “But you aren’t in/on the X”, if appropriate. [L61017]
- The response to READ was inappropriate when an object is mispelled or out of scope. [L61016]
- A small bug in the choice of library messages for PUSH and TURN, which wasn’t noticeable unless you overrode the messages to be different from PULL, has been corrected. [L61015]
- If you are in a dark room, you cannot examine what you are holding. Yet if you open a container you brought in from a lit room, the standard message “You open the box, revealing a...” was not being suppressed. [L61014]
- The `ScoreMatchL()` routine in Parserm.h incorrectly decided which objects meet descriptors. As a result, some objects that didn’t meet descriptors were not properly removed from the match list when the library is deciding which objects best match a player’s input. [L61013]
- The Infix problem parsing commands containing commas has been, erm, fixed. [L61010]
- A problem when describing what’s visible after opening a container has been corrected. [L61009]
- An inappropriate message after GO NORTH CIRCULAR has been corrected. [L61008]
- Modified foreground and background colours are now correct after RESTORE and UNDO. [L61007]
- The `grammar` property now works with a large game whose dictionary lies above $8000. [L61006]
- A buffer conflict with disambiguation and UNDO has been resolved. [L61004]
- If a player is inside a closed, non-transparent container, the library prints an extra blank line between the header “The container” and the first `inside_description` line it prints. No more. [L61002]
- The list writing routines do not handle plural containers correctly. If you have two empty boxes, it might list “two boxes (which is closed)”. Not only should it say “are closed”, but it will lump empty containers together even if some are open and others aren’t. Now resolved. [L61001]
- A conflict between `DrawStatusLine()` and `DisplayStatus()` on how to determine whether to display turns or time is settled in favour of checking a header flag. [L60709]

Support for Glulx

One of the limitations of the Z-machine is the size of the largest game that it supports: 256Kb if you compile a version 5 game, or 512Kb if you compile for version 8. If you find yourself up against this limit and you've tried all the standard tricks to save a few bytes here and there, then it's time to switch to Glulx. That's easy to do: you just supply the **-G** compiler switch, the compiler generates Glulx code, and any Glulx interpreter will be able to run it.

Actually, it isn't always quite that simple. See:

- "Knowing which is which" below
- "Glulx differences" on page 14
- "Glulx additions" on page 17

Knowing which is which

As mentioned earlier, the compiler automatically defines a couple of useful constants. If you're compiling for the Z-machine then `TARGET_ZCODE` is defined; if you're compiling for Glulx then you're given `TARGET_GLULX` instead. You can use these with the `Ifdef` directive, like this:

```
#Ifdef TARGET_ZCODE;
!   Z-machine code here
#Endif;

#Ifdef TARGET_GLULX;
!   Equivalent Glulx code here
#Endif;
```

or more commonly like this:

```
#Ifdef TARGET_ZCODE;
!   Z-machine code here
#Ifnot;
!   Equivalent Glulx code here
#Endif;
```

You'll find a lot of this if you look in the library files, but it's less frequently needed in a source file.

Glulx differences

The two VMs are not identical, and you need to be aware of their differences. See:

Word size

The most basic difference between Glulx and the Z-machine is that words are four bytes long instead of two. All Glulx variables are 32-bit values, the stack contains 32-bit values, property entries are 32-bit values, and so on.

In most Inform programming, you don't need to worry about this change at all. For example, if you have an array

```
Array mylist --> 10;
```

...then Z-code Inform allocates ten words – that is, twenty bytes – and you can access these values as `mylist-->0` through `mylist-->9`. If you compile the same code under Glulx Inform, the compiler again allocates ten words – now forty bytes – and you can still access them as `mylist-->0` through `mylist-->9`. Everything works the same, except that the array can contain values greater than the Z-machine's limit of 65535.

`Table` arrays also refer to two- or four-byte word values, and the first word is the length of the array. `String` and `->` arrays, and the `->` notation, still refer to single bytes. You should not have to modify your code here, either.

There are two important cases where you **will** have to modify your code. First is the `.#` operator. The expression *`obj`*`.#`*`prop`* returns the length of the property in bytes. Since properties almost always contain words, rather than bytes, it is very common to have Z-machine code like:

```
len = (obj.#prop) / 2;
for (i=0 : i<len : i++)
    print (string) (obj.&prop)-->i;
```

In Glulx Inform programs, it is necessary to divide by 4 instead of by 2, so you should replace the above code with:

```
len = (obj.#prop) / WORDSIZE;
for (i=0 : i<len : i++)
    print (string) (obj.&prop)-->i;
```

This will compile and run correctly in both VMs.

The other circumstance where your code may need modifying in this manner is when using the 'print to array' feature. Code like this:

```
Array mybuf -> 100; ! must be big enough for largest string you'll print
mystr = "hello";
mystr.print_to_array(mybuf);
```

results in the first **word** of `mybuf` containing 5 (the number of characters in `mystr`), and the following five **bytes** containing 'h', 'e', 'l', 'l' and 'o'. In the Z-machine, you could then output the characters from the array with either of these code fragments:

```
len = 2 + mybuf-->0
for (i=2 : i<len : i++)
    print (char) mybuf->i;
len = mybuf-->0
for (i=0 : i<len : i++)
    print (char) mybuf->(i+2);
```

Again, you can make the code safe for both VMs if you change "2" to "WORDSIZE":

```
len = WORDSIZE + mybuf-->0
for (i=WORDSIZE : i<len : i++)
    print (char) mybuf->i;
len = mybuf-->0
for (i=0 : i<len : i++)
    print (char) mybuf->(i+WORDSIZE);
```

See also "Features available only in Glulx" on page 17 for an extended form of print_to_array.

Directives

Glulx handles the majority of Inform directives, with two exceptions:

- The Zcharacter directive causes the compilation error "Glulx Inform does not handle Unicode yet". The message, though true, is misleading; the real issue is that Glulx does not use the ZSCII character set, which can in part be configured by various forms of Zcharacter. Your best approach is to bypass the directive when compiling for Glulx:

  ```
  #Ifdef TARGET_ZCODE;
  Zcharacter ... ;
  #Endif;
  ```

- The (obsolete) Lowstring directive causes a run-time error when used like this:

  ```
  Lowstring mystr "hello";

  string 0 mystr;
  print "@00 and goodbye.";
  ```

  This simpler form, avoiding the use of Lowstring, works successfully:

  ```
  string 0 "hello";
  print "@00 and goodbye.";
  ```

  See also "Features available only in Glulx" on page 17 for additional printing variable support.

Statements

Glulx handles the majority of Inform statements, with a few exceptions. If you try to use any of these statements in Glulx, you will cause a compilation error:

- save, restore: These are more complicated procedures in Glulx than in Z-code, and cannot be implemented without involving library variables and routines. If you want to do this sort of thing, modify or copy the library SaveSub() and RestoreSub() routines.
- read: Similarly, reading a line of text in Glulx involves the library; the compiler cannot generate stand-alone code to do it. See instead the library KeyboardPrimitive() routine.
- @*opcode*: The Z-machine assembly language is completely different from that of Glulx. If you have used any assembly instructions, you will need to conceal them when compiling for Glulx, as

described in “Knowing which is which” on page 13. See also “Features available only in Glulx” on page 17 for details of the `glk()` function call.

Character handling

Unlike the Z-machine, which internally uses the ZSCII character set (see the *Inform Designer's Manual* Table 2 on p. 519), Glulx sticks to the ISO 8859-1 (Latin-1) encoding. This eight-bit scheme is the same as ZSCII for character values 0-127, but different for character values 128-255. Note that Glulx is built on the Glk I/O library, which currently does not handle Unicode characters (except insofar as code points $0000 through $00FF are identical to ISO 8859). The impact is as follows:

- `@`*`escape_sequence`*: Escape sequences such as `@:a` and `@LL` (for “ä” and “£” respectively) are accepted identically by both VMs (except that Glulx does not handle `@oe` and `@OE`, because the “oe” and “OE” ligatures are outside the range of ISO 8859-1).
- `@@`*`decnum`*: The number is the character's internal decimal value, so `@@65` is “A” in both VMs, but `@@165` is “ï” in the Z-machine and “¥” in Glulx.
- `@{`*`hexnum`*`}`: The number is the character's Unicode value, so `@{41}` is “A” and `@{EF}` is “ï” in both VMs. However, `@{A5}` is “¥” in Glulx, but causes a Z-machine compilation error because “¥” isn't a ZSCII character.
- `-C`*`n`*: The compiler switches **`-C1`** through **`-C9`** specify that the source file uses the character set defined by ISO 8859-1 through 8859-9 respectively. In the Z-machine, the switch also initialises the higher ZSCII character set to appropriate values; this feature is irrelevant when compiling for Glulx.

Glulx additions

Glulx offers some additional capabilities above those of the Z-machine. See:

- "Features available only in Glulx" below
- "Library routines available only in Glulx" on page 18
- "Entry points available only in Glulx" on page 20

Features available only in Glulx

- There is an extended form of `print_to_array`:

  ```
  len = mystr.print_to_array(mybuf, 80);
  ```

  This example writes no more than 76 characters into the array. If *`mybuf`* is an 80-byte array, you can be sure it will not be overrun. (Do not try this with the second argument less than 4.)

  The value written into `mybuf-->0`, and the value returned, are **not** limited to the number of characters written; they represent the number of characters in the complete string. This means that:

  ```
  len = mystr.print_to_array(mybuf, 4);
  ```

  is an ugly but perfectly legal way to find the length of a string. (And in this case, `mybuf` need only be four bytes long.)

- Z-code Inform supports 32 printing variables, `@00` to `@31`, which you can include in strings and then set with the statement:

  ```
  string num "value";
  ```

  In Glulx, this limit is raised to 64. Furthermore, in Glulx you can set these variables to a stand-alone routine as well as a string:

  ```
  [ routine; print "value"; ];

  string num routine;
  ```

  In this case, the routine is called with no arguments and the result discarded; you should print your desired output inside the routine.

  In Glulx, unlike Z-code, a printing variable string can itself contain `@..` codes, allowing recursion. You can nest this as deeply as you want. However, it is obviously a bad idea to cause an infinite recursion. For example, this will certainly crash the interpreter:

  ```
  string 3 "This is a @03!";
  print "What is @03?";
  ```

- Many of the things that used to be Z-code assembly are now handled by Glk function calls. Making a Glk function call from Inform is slightly screwy, but not difficult.

  All of Glk is handled by the built-in Inform function `glk()`, which takes one or more arguments. The first argument is an integer; this tells **which** Glk call in being invoked. The remaining arguments are just the arguments to the Glk call, in order.

  Say, for example, that you want to set the text style to "preformatted". The Inform code to accomplish this is:

  ```
  glk($0086, 2);
  ```

The hex value $0086 means `glk_set_style`; the value "2" means `Preformatted`.

The table of Glk calls, and the integers that refer to them, is in the Glk specification (remember that the values given there are hexadecimal). See:

```
http://www.eblong.com/zarf/glk/glk-spec-061_11.html#s.1.6
```

Since calls based on numeric codes are not very easy to read, we recommend that you download John Cater's `infglk.h` library header, which defines wrapper functions and constants. For example, you could achieve the same effect with this call:

```
glk_set_style(style_Preformatted);
```

When you read the Glk specification, bear in mind that the NULL value it talks about is the C language `NULL` (0), not the Inform Library `NULL` (-1). `infglk.h` defines a constant `GLK_NULL` (equal to 0) which you can use where appropriate.

- By default, arguments to routines work the same in Glulx as they do in Z-code. When you call a routine, the arguments that you pass are written into the routine's local variables, in order. If you pass too many arguments, the extras are discarded; too few, and the unused local variables are filled with zeroes.

  However, the Glulx VM supports a second style of routine. You can define a routine of this type by naming the first argument `_vararg_count`. For example:

  ```
  [ StackFunc _vararg_count ix pos len;
      !   Glulx code here
  ];
  ```

  If you do this, the routine arguments are **not** written into the local variables. Instead, they are pushed onto the stack, and you must use Glulx assembly to pull them off. All the local variables are initialized to zero, except for `_vararg_count`, which (as you might expect) contains the number of arguments that were passed in.

  Note that `_vararg_count` is a normal local variable, aside from its useful initial value. You can assign to it, increment or decrement it, use it in expressions, and so on.

  Stack-argument routines are most useful if you want a routine with variable arguments, or if you want to write a wrapper that passes its arguments on to another routine.

Library routines available only in Glulx

`KeyCharPrimitive(`*`win`*`, `*`nostat`*`);`

If *`win`* is nonzero, the character input request goes to that Glk window (instead of `gg_mainwin`, the default.) If *`nostat`* is nonzero, a window rearrangement event is returned immediately as value 80000000 (instead of the default behavior, which is to call `DrawStatusLine()` and keep waiting.)

`PrintAnything(`*`thingie`*`, ...);`

In the Z-machine, strings and routines are "packed" addresses, dictionary words are normal addresses, and game objects are represented as sequential numbers from 1 to `#top_object`. These ranges overlap; a string, a dictionary word, and an object could conceivably all be represented by the same numeric value.

In Glulx, all those things are represented by normal addresses, so different items will always have different values. Furthermore, the first byte found at the address is an identifier value, which specifies what kind of item the address contains.

`PrintAnything()` prints any *`thingie`* – string, routine (with optional arguments), object, object property (with optional arguments), or dictionary word – known to the library.

Calling	**Is equivalent to**
`PrintAnything()`	(nothing printed)
`PrintAnything(0)`	(nothing printed)
`PrintAnything(`*`string`*`)`	`print (string) "`*`string`*`";`
`PrintAnything(`*`dictionaryword`*`)`	`print (address) '`*`dictionaryword`*`';`
`PrintAnything(`*`obj`*`)`	`print (name)` *`obj`*`;`
`PrintAnything(`*`obj`*`, `*`prop`*`)`	*`obj.prop`*`();`
`PrintAnything(`*`obj`*`, `*`prop`*`, `*`args...`*`)`	*`obj.prop`*`(`*`args...`*`);`
`PrintAnything(`*`routine`*`)`	*`routine`*`();`
`PrintAnything(`*`routine`*`, `*`args...`*`)`	*`routine`*`(`*`args...`*`);`

Extra arguments after a *`string`* or dictionary *`word`* are safely ignored.

The (first) argument you pass in is always interpreted as a thingie reference, not as an integer. This is why none of the forms shown above print out an integer. However, you can get the same effect by calling

```
PrintAnything(DecimalNumber, num);
```

...which is where the `DecimalNumber()` routine comes in handy. You can also, of course, use other library routines, and do tricks like

```
PrintAnything(EnglishNumber, num);
PrintAnything(DefArt, obj);
```

None of this may seem very useful; after all, there are already ways to print all those things. But `PrintAnything()` is vital in implementing the following routine:

```
PrintAnyToArray(array, arraylen, thingie, ...);
```

This works the same way, except that instead of printing to the screen, the output is diverted to the given array.

The first two arguments must be the array address and its maximum length. Up to that many characters will be written into the array; any extras will be silently discarded. This means that you do not have to worry about array overruns.

The `PrintAnyToArray()` routine returns the number of characters generated. (This may be greater than the length of the array. It represents the entire text that was output, not the limited number written into the array.)

It is safe to nest `PrintAnyToArray()` calls. That is, you can call `PrintAnyToArray(`*`routine`*`)`, where *`routine`*`()` itself calls `PrintAnyToArray()`. (However, if they try to write to the **same array**, chaos will ensue.)

It is legal for *`arraylen`* to be zero (in which case *`array`* is ignored, and may be zero as well.) This discards **all** of the output, and simply returns the number of characters generated. You can use this to find the length of anything – even a function call.

Entry points available only in Glulx

An entry point is a routine which you can provide in your code, or leave out; the library will call it if it's present, ignore it if not – see §21 of the *Inform Designer's Manual*.

The library has some entry points which aid in writing more complicated interfaces – games with sound, graphics, extra windows, and other fancy Glk tricks. If you're just writing a standard Infocom-style game, **you can ignore this section**.

`HandleGlkEvent(`*`ev`*`, `*`context`*`, `*`abortres`*`)`

This entry point is called every time a Glk event occurs. The event could indicate nearly anything: a line of input from the player, a window resize or redraw event, a clock tick, a mouse click, or so on.

The library handles all the events necessary for a normal Infocom-style game. You need to supply a `HandleGlkEvent()` routine only if you want to add extra functionality. The *`ev`* argument is a four-word array which describes the event. *`ev`*`-->0` is the type of the event; *`ev`*`-->1` is the window involved (if relevant); and *`ev`*`-->2` and *`ev`*`-->3` are extra information. The *`context`* argument is 0 if the event occurred during line input (normal commands, `YesOrNo()`, or some other use of the `KeyboardPrimitive()` library routine); 1 indicates that the event occurred during character input (any use of the `KeyCharPrimitive()` library routine). The *`abortres`* argument is used only if you want to cancel player input and force a particular result; see below.

If you return 2 from `HandleGlkEvent()`, player input will immediately be aborted. Some additional code is also required:

- If this was character input (*`context`*`==1`), you must call the Glk `cancel_char_event` function, and then set *`abortres`*`-->0` to the character you want returned. Then return 2; `KeyCharPrimitive()` will end and return the character, as if the player had hit it.
- If this was line input (*`context`*`==0`), you must call the Glk `cancel_line_event` function. (You can pass an array argument to see what the player had typed so far.) Then, fill in the length of the input to be returned in *`abortres`*`-->0`. If this is nonzero, write the input characters sequentially into the array starting at *`abortres`*`->WORDSIZE`, up to (but not including) *`abortres`*`->(WORDSIZE+len)`. Do not exceed 256 characters. Then return 2; `KeyboardPrimitive()` will end and return the line.

If you return -1 from `HandleGlkEvent()`, player input will continue even after a keystroke (for character input) or after the enter key (for line input). (I don't know why this is useful, but it might be.) You must re-request input by calling `request_char_input` or `request_line_input`.

Any other return value from `HandleGlkEvent()` (a normal return, `rfalse`, or `rtrue`) will not affect the course of player input.

`InitGlkWindow(`*winrock*`)`

This entry point is called by the library when it sets up the standard windows: the story window, the status window, and (if you use quote boxes) the quote box window. The story and status windows are created when the game starts (before `Initialise()`). The quote window is created and destroyed as necessary.

`InitGlkWindow()` is called in five phases:

1. The library calls `InitGlkWindow(0)`. This occurs at the very beginning of execution, even before `Initialise()`. You can set up any situation you want. (However, remember that the story and status windows might already exist – for example, if the player has just typed RESTART.) This is a good time to set `gg_statuswin_size` to a value other than 1. Return 0 to proceed with the standard library window setup, or 1 if you've created all the windows yourself.
2. The library calls `InitGlkWindow(GG_MAINWIN_ROCK)`, before creating the story window. This is a good time to set up style hints for the story window. Return 0 to let the library create the window; return 1 if you have yourself created a window and stored it in `gg_mainwin`.
3. The library calls `InitGlkWindow(GG_STATUSWIN_ROCK)`, before creating the status window. Again, return 0 to let the library do it; return 1 if you have created a window and stored it in `gg_statuswin`.
4. The library calls `InitGlkWindow(1)`. This is the end of window setup; you can take this opportunity to open other windows. (Or you can do that in your `Initialise()` routine. It doesn't matter much.)
5. The library calls `InitGlkWindow(GG_QUOTEWIN_ROCK)`, before creating the quote box window. This does not occur during game initialization; the quote box window is created during the game, whenever you print a quote, and destroyed one turn later. As usual, return 1 to indicate that you've created a window in `gg_quotewin`. (The desired number of lines for the window can be found in `gg_arguments-->0`.)

However you handle window initialization, remember that the library requires a `gg_mainwin`. If you don't create one, and don't allow the library to do so, the game will shut down. Contrariwise, the status window and quote windows are optional; the library can get along without them.

`IdentifyGlkObject(`*`phase`*`, `*`type`*`, `*`ref`*`, `*`rock`*`)`

This entry point is called by the library to let you know what Glk objects exist. You must supply this routine if you create any windows, filerefs, file streams, or sound channels beyond the standard library ones. (This is necessary because after a RESTORE, RESTART, or UNDO command, your global variables containing Glk objects will be wrong.)

`IdentifyGlkObject()` is called in three phases:

1. The library calls `IdentifyGlkObject()` with *`phase`*`==0`. You should set all your Glk object references to zero.
2. The library calls `IdentifyGlkObject()` with `phase==1`. This occurs once for each window, stream, and fileref that the library doesn't recognize. (The library handles the two standard windows, and the files and streams that have to do with saving, transcripts, and command records. You only have to deal with objects that you create.) You should set whatever reference is appropriate to the object. For each object: *`type`* will be 0, 1, 2 for windows, streams, filerefs respectively; *`ref`* will be the object reference; and *`rock`* will be the object's rock, by which you can recognize it.
3. The library calls `IdentifyGlkObject()` with `phase==2`. This occurs once, after all the other calls, and gives you a chance to recognize objects that aren't windows, streams, or filerefs. If you don't create any such objects, you can ignore that bit. But you should also take the opportunity to update all your Glk objects to the game state that was just started or restored. (For example, redraw graphics, or set the right background sounds playing.)