The cblorb Manual P/man

Purpose

A guide for users of cblorb.

---

§**1. Some definitions.** `cblorb` is a command-line tool which forms one of the components of the Inform 7 design system for interactive fiction. All installations of Inform 7 contain a copy of `cblorb`, though few users are aware of it, since it doesn't usually communicate with them directly. Instead, the Inform user interface calls it when needed. The moment comes at the end of the translation process, but only when the Release button rather than the Go or Replay buttons was clicked. `cblorb` has two main jobs: to bind up the translated project, together with any pictures, sounds, or cover art, into a single file called a "blorb" which can be given to players on other machines to play; and to produce associated websites, solution files and so on as demanded by "Release..." instruction(s) in the source text.

§**2.** "Blorb" is a general-purpose wrapper format designed as a way to gather together audiovisual media and bibliographic data for works of IF. The format was devised and formally specified by Andrew Plotkin around 2000, and its name is borrowed from that of a magic spell in Infocom's classic work, *Enchanter*. ("The blorb spell (safely protect a small object as though in a strong box).") Although Inform 6, the then current version, did not itself generate blorb files, a Perl script called `perlblorb` was provided in 2001 so that the user could perform the wrapping-up process. `perlblorb` is no longer used, and survives only in the name of `cblorb`, which is a C version of what had previously been written in Perl. This means it can run on machines with no Perl installation, which Inform 7 needs to be able to do. Unlike `perlblorb`, `cblorb` is "under the hood"; the user does not need to give it instructions. This manual is therefore useful only for people needing to generate Inform-related websites, or who are maintaining the Inform user interface applications.

§**3.** Sentences in Inform source text such as:

> Release along with public source text, cover art, and a website.

do in effect transmit instructions to `cblorb`, but `cblorb` doesn't read them in this natural-language form. Instead, the `ni` component of Inform 7 translates these instructions into a script for `cblorb` to follow. This script is called a "blurb".

"Blurb" is a mini-language for specifying how the materials in a work of IF should be packaged up for release. It was originally codified in 2001 as a standard way to describe how a blorb file should be put together, but it was extended in 2005 and again in 2008 so that it could also organise accompanying files released along with the blorb.

The original Blurb language was documented in chapter 43 of the DM4 (i.e., the *Inform Designer's Manual*, fourth edition, 2001); for clarity, we will call that language "Blurb 2001". Today's Blurb language is a little different. Some features of Blurb 2001 are deprecated and no longer used, while numerous other syntaxes are new. Because of this the DM4 specification is no longer useful, so we will give a full description below of Blurb as it currently stands.

§**4.** `cblorb` **within the Inform user interface.** This is the sequence of events when the user clicks Release in the user interface application (the "interface"):

(1) The interface calls `ni`, the I7 compiler, as normal except that the `-release` command-line switch is specified.

(2) `ni` compiles the source text into I6 code. If Problems occur, `ni` exits with a return code of 1, and the interface displays those, and then stops the process.

(3) If no Problems occur, `ni` writes two additional files besides the I6 code it always writes:
   (a) `Metadata.iFiction`, an iFiction record;
   (b) `Release.blurb`, a blurb file of instructions for `cblorb` to follow later.

(4) `ni` having returned 0 to indicate success, the interface next calls the Inform 6 compiler (called, e.g., `inform-6.31-biplatform`, but we'll call it `i6` here). The interface calls `i6` as normal except that the `S` and `D` switches, for strict checking and for debugging, are off instead of on. If `ni` works properly then `i6` should certainly not produce syntax errors, though it will surely produce warnings; all the same it can fail if, say, Z-machine memory limits are exceeded. The interface should deal with such failures exactly as it would in a non-Release run.

(5) `i6` having returned 0 to indicate success, the interface next calls `cblorb` as follows. Let `Path` be the path to the folder containing the Inform project being released, which we'll call `This.inform`. Then the interface should call:

```
cblorb -platform "Path/This.inform/Release.blurb" "Path/This.inform/Build/output.gblorb"
```

where `-platform` should be one of `-osx`, `-windows` or `-unix`. (The default is `-osx`.) The two filename arguments are the Blurb script for `cblorb` to follow, which was written by `ni` at step 3, and the filename of the Blorb file which it should write. Note that the interface should give this the extension ".gblorb" if the Glulx setting is in force, and ".zblorb" if the Z-machine.

(6) Like its predecessors, `cblorb` can produce error messages, and it returns 0 (success) or 1 (failure). But the interface doesn't actually need to look at that, because `cblorb` also produces a much fuller report in the form of an HTML page to be displayed on the Problems tab. This is `StatusCblorb.html`, in the project's `Build` folder. (This is a change made in 2010: in the past, the interface simply chose between a generic success and failure page on the basis of the return code.)

(7) There are no more tools to call, but the interface has one last duty (if `cblorb` succeeded) – to move the blorb somewhere sensible on disc, where the user can see it. Leaving it where it is will not do – the user never looks inside the Build project of a folder, which on Mac OS X, for instance, is not even visible. To see what to do, the interface must look at the textual output from `cblorb`, printed to `stdout` (of course the interface is free to redirect this if it wants to). If `cblorb` printed a line in the form:

```
Copy blorb to: [[...]]
```

then the interface should do as it's told. For instance:

```
Copy blorb to: [[/Users/gnelson/Examples/Bronze Materials/Release/Bronze.gblorb]]
```

If `cblorb` printed no such line, the interface should put up a Save As... dialogue box, and invite the user to choose a destination.

§**5.** `cblorb` **at the command line.** When using `cblorb` as a command-line tool, it's probably convenient to download a standalone copy from the Inform website, though that's identical to the copy squirreled away somewhere in the application. On Mac OS X, it lives at:

```
Inform.app/Contents/Resources/Compilers/cBlorb
```

Its main usage is:

```
cblorb -platform [-options] blurbfile [blorbfile]
```

where `-platform` should be one of `-osx`, `-windows`, `-unix`. At present the only practical difference this makes is that the Windows setting causes `cblorb` to use `\` instead of `/` as a filename separator.

The blorbfile filename is optional since `cblorb` does not always need to make a blorb; that depends on the instructions handed to it in the `blurbfile`.

§**6.** The other command-line options are:

`-help`: prints summaries of command-line use and the Blurb language.

`-trace`: mainly for debugging, but possibly also useful as a verbose mode.

`-project Whatever.inform`: tells `cblorb` to assume the usual settings for this project. (That means the blurbfile is set to `Whatever.inform/Release.blurb` and the blorbfile to `Whatever.inform/Build/output.gblorb`.)

§**7. Example blurb scripts.** This first script instructs `cblorb` to carry out its mission – it makes a simple Blorb wrapping up a story file with bibliographic data, but nothing more, and nothing else is released.

```
storyfile "/Users/gnelson/Examples/Zinc.inform/Build/output.ulx" include
ifiction "/Users/gnelson/Examples/Zinc.inform/Metadata.iFiction" include
```

These two lines tell `cblorb` to include the story file and the iFiction record respectively.

§**8.** A more ambitious Blorb can be made like so:

```
storyfile leafname "Audiophilia.gblorb"
storyfile "/Users/gnelson/Examples/Audiophilia.inform/Build/output.ulx" include
ifiction "/Users/gnelson/Examples/Audiophilia.inform/Metadata.iFiction" include
cover "/Users/gnelson/Examples/Audiophilia Materials/Cover.png"
picture 1 "/Users/gnelson/Examples/Audiophilia Materials/Cover.png"
sound 3 "/Users/gnelson/Examples/Audiophilia Materials/Sounds/Powermac.aiff"
sound 4 "/Users/gnelson/Examples/Audiophilia Materials/Sounds/Bach.ogg"
```

The cover image is included only once, but declaring it as picture 1 makes it available to the story file for display internally as well as externally. Resource ID 2, apparently skipped, is in fact the story file.

§**9.** And here's a very short script, which makes `cblorb` generate a solution file from the Skein of a project:

```
project folder "/Users/gnelson/Examples/Zinc.inform"
release to "/Users/gnelson/Examples/Zinc Materials/Release"
solution
```

This time no blorb file is made. The opening line tells `cblorb` which Inform project we're dealing with, allowing it to look at the various files inside – its Skein, for instance, which is used to create a solution. The second line tells `cblorb` where to put all of its output – everything it makes. Only the third line directly causes `cblorb` to do anything.

§**10.** More ambitiously, this time we'll make a website for a project, but again without making a blorb:

```
project folder "/Users/gnelson/Examples/Audiophilia.inform"
release to "/Users/gnelson/Examples/Audiophilia Materials/Release"
placeholder [IFID] = "AD5648BA-18A2-48A6-9554-4F6C53484824"
placeholder [RELEASE] = "1"
placeholder [YEAR] = "2009"
placeholder [TITLE] = "Audiophilia"
placeholder [AUTHOR] = "Graham Nelson"
placeholder [BLURB] = "A test project for sound effect production."
template path "/Users/gnelson/Library/Inform/Templates"
css
website "Standard"
```

The first novelty here is the setting of placeholders. These are named pieces of text which appear on the website being generated: where the text "[RELEASE]" appears in the template, `cblorb` writes the value we've set for it, in this case "1". Some of these values look like numbers, but to `cblorb` they all hold text. A few placeholder names are reserved by `cblorb` for its own use, and it will produce errors if we try to set those, but none of those in this example is reserved.

Template paths tell `cblorb` where to find templates. Any number of these can be set – including none at all, but if so then commands needing a named template, like `website`, can't be used. `cblorb` looks for any template it needs by trying each template path in turn (the earliest defined having the highest priority). The blurb files produced by `ni` in its `-release` mode containa chain of three template paths, for the individual project folder, the user's library of installed templates, and the built-in stock inside the Inform user interface application, respectively.

The command `css` tells `cblorb` that it is allowed to use CSS styles to make its web pages more appealing to look at: this results in generally better HTML, easier to use in other contexts, too.

All of that set things up so that the `website` command could be used, which actually does something – it creates a website in the release-to location, taking its design from the template named. If we were to add any of these commands –

```
source public
solution public
ifiction public
```

– then the website would be graced with these additions.

§**11.** The previous examples all involved Inform projects, but `cblorb` can also deal with stand-alone files of Inform source text – notably extensions. For example, here we make a website out of an extension:

```
release to "Test Site"
placeholder [TITLE] = "Locksmith"
placeholder [AUTHOR] = "Emily Short"
placeholder [RUBRIC] = "Implicit handling of doors and..." and so on
template path "/Users/gnelson/Library/Inform/Templates"
css
release file "style.css" from "Extended"
release file "index.html" from "Extended"
release file "Extensions/Emily Short/Locksmith.i7x"
release source "Extensions/Emily Short/Locksmith.i7x" using "extsrc.html" from "Extended"
```

This time we're using a template called "Extended", and the script tells `cblorb` exactly what to do with it. The "release file... from..." command tells `cblorb` to extract the named file from this template and to copy it into the release folder – if it's a ".html" file, placeholders are substituted with their values. The simpler form, "release file ...", just tells `cblorb` to copy that actual file – here, it puts a copy of the extension itself into the release folder. The final line produces a run of pages, in all likelihood, for the source and documentation of the extension, with the design drawn from "Extended" again.

("Extended" isn't supplied inside Inform; it's a template we're using to help generate the Inform website, rather than something meant for end users. There's nothing very special about it, in any case.)

§**12. Specification of the Blurb language.** A blorb script should be a text file, using the Unicode character set and encoded as UTF-8 without a byte order marker – in other words, a plain text file. It consists of lines of up to 10239 bytes in length each, divided by any of the four line-end markers in common use (`CR`, `LF`, `CR LF` or `LF CR`), though the same line-end marker should be used throughout the file.

Each command occupies one and only one line of text. (In Blorb 2001, the now-deprecated `palette` command could occupy multiple lines, but `cblorb` will choke on such a usage.) 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. An entirely empty blurb file, containing nothing but white space, is perfectly legal though useless.

In the following description:

⟨*string*⟩ means any text within double-quotes, not containing either double-quote or new-line characters, of up to 2048 bytes.

⟨*filename*⟩ means any double-quoted filename.

⟨*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.

§**13.** The full set of commands is as follows. First, core commands for making a blorb:

`author` ⟨*string*⟩

Adds this author name to the file.

`copyright` ⟨*string*⟩

Adds this copyright declaration to the blorb file. It would normally consist of short text such as "(c) J. Mango Pineapple 2007" rather than a lengthy legal discourse.

`release` ⟨*number*⟩

Gives this release number to the blorb file.

`auxiliary` ⟨*filename*⟩ ⟨*string*⟩

Tells us that an auxiliary file – for instance, a PDF manual – is associated with the release but will not be embedded directly into the blorb file. For instance,

```
auxiliary "map.png" "Black Pete's treasure map"
```

The string should be a textual description of the contents. Every auxiliary file should have a filename including an extension usefully describing its format, as in ".png": if there is no extension, then the auxiliary resource is assumed to be a mini-website housed in a subfolder with this name.

`ifiction` ⟨*filename*⟩ `include`

The file should be a valid iFiction record for the work. This is an XML file specified in the Treaty of Babel, a cross-IF-system standard for specifying bibliographic data; it will be embedded into the blorb.

`storyfile` ⟨*filename*⟩ *unsupported by* `cblorb`
`storyfile` ⟨*filename*⟩ `include`

Specifies the filename of the story file which these resources are being attached to. Blorb 2001 allowed for blorbs to be made which held everything to do with the release *except* the story file; that way a release might consist of one story file plus one Blorb file containing its pictures and sounds. The Blorb file would then contain a note of the release number, serial code and checksum of the associated story file so that an interpreter can try to match up the two files at run-time. 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. `cblorb` always does this, and does not support `storyfile` without `include`.

§**14.** Second, now-deprecated commands describing our ideal screen display:

`palette 16 bit` — *unsupported by* `cblorb`
`palette 32 bit` — *unsupported by* `cblorb`
`palette {` ⟨*colour-1*⟩ ⟨*colour-N*⟩ `}` — *unsupported by* `cblorb`

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*⟩ — *unsupported by* `cblorb`
`resolution` ⟨*dim*⟩ `min` ⟨*dim*⟩ — *unsupported by* `cblorb`
`resolution` ⟨*dim*⟩ `max` ⟨*dim*⟩ — *unsupported by* `cblorb`
`resolution` ⟨*dim*⟩ `min` ⟨*dim*⟩ `max` ⟨*dim*⟩ — *unsupported by* `cblorb`

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 0×0 and $\infty \times \infty$.

§**15.** Third, commands for adding audiovisual resources:

`sound` ⟨*id*⟩ ⟨*filename*⟩
`sound` ⟨*id*⟩ ⟨*filename*⟩ `repeat` ⟨*number*⟩ — *unsupported by* `cblorb`
`sound` ⟨*id*⟩ ⟨*filename*⟩ `repeat forever` — *unsupported by* `cblorb`
`sound` ⟨*id*⟩ ⟨*filename*⟩ `music` — *unsupported by* `cblorb`
`sound` ⟨*id*⟩ ⟨*filename*⟩ `song` — *unsupported by* `cblorb`

Tells us to take a sound sample from the named file and make it the sound effect with the given number. Most forms of `sound` are now deprecated: repeat information (the number of repeats to be played) is meaningful only with Z-machine version 3 story files using sound effects, and Inform 7 does not generate those; the `music` and `song` keywords specify unusual sound formats. Nowadays the straight `sound` command should always be used regardless of format.

`picture` ⟨*id*⟩ ⟨*filename*⟩
`picture` ⟨*id*⟩ ⟨*filename*⟩ `scale` ⟨*ratio*⟩ — *unsupported by* `cblorb`
`picture` ⟨*id*⟩ ⟨*filename*⟩ `scale min` ⟨*ratio*⟩ — *unsupported by* `cblorb`
`picture` ⟨*id*⟩ ⟨*filename*⟩ `scale` ⟨*ratio*⟩ `min` ⟨*ratio*⟩ — *unsupported by* `cblorb`

(and so on) is a similar command for images. In 2001, the image file was required to be a PNG, but it can now alternatively be a JPEG.

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".

`cblorb` does not support any of the scaled forms of `picture`. As with the exotic forms of `sound`, they now seem passé. We no longer need to worry too much about the size of the blorb file, nor about screens with very low resolution; an iPhone today has a screen resolution close to that of a typical desktop of 2001.

`cover` ⟨*filename*⟩

specifies that this is the cover art; it must also be declared with a `picture` command in the usual way, and must have picture ID 1.

§**16.** Three commands help us to specify locations.

`project folder` ⟨*filename*⟩

Tells `cblorb` to look for associated resources, such as the Skein file, within this Inform project.

`release to` ⟨*filename*⟩

Tells `cblorb` that all of its output should go into this folder. (Well, except that the blorb file itself will be written to the location specified in the command line arguments, but see the description above of how `cblorb` then contrives to move it.) The folder must already exist, and `cblorb` won't create it. Under some circumstances Inform will seem to be creating the release folder if it doesn't already exist, but that's always the work of `ni`, not `cblorb`.

`template path` ⟨*filename*⟩

Sets a search path for templates – a folder in which to look for them. There can be any number of template paths set, and `cblorb` checks them in order of declaration (i.e., most important first).

§**17.** Next we come to commands for specifying what `cblorb` should release. At present it has seven forms of output: Blorb file, solution file, source text, iFiction record, miscellaneous file, website and interpreter.

No explicit single command causes a Blorb file to be generated; it will be made automatically if one of the above commands to include the story file, pictures, etc., is present in the script, and otherwise not generated.

```
solution
solution public
```

causes a solution file to be generated in the release folder. The mechanism for this is described in *Writing with Inform*. The difference between the two commands affects only a website also being made, if one is: a public solution will be included in its links, thus being made available to the public who read the website.

```
ifiction
ifiction public
```

is similar, but for the iFiction record of the project.

```
source
source public
```

is again similar, but here there's a twist. If the source is public, then `cblorb` doesn't just include it on a website: it generates multiple HTML pages to show it off in HTML form, as well as including the plain text original.

Miscellaneous files can be released like so:

`release file` ⟨*filename*⟩

Here `cblorb` acts as no more than a file-copy utility; a verbatim copy of the named file is placed in the release folder.

§**18.** Finally we come to web pages.

`css`

enables the use of CSS-defined styles within the HTML generated by `cblorb`. This has an especially marked effect when `cblorb` is generating HTML versions of Inform source text, and is *a good thing*. Unless there is reason not to, every blurb script generating websites ought to contain this command.

`release file` ⟨*filename*⟩ `from` ⟨*template*⟩

causes the named file to be found from the given template. If it can't be found in that template, `cblorb` tries to find it from a template called "Standard". If it isn't there either, or `cblorb` can't find any template called "Standard" in any of its template paths (see above), then an error message is produced. But if all goes well the file is copied into the release folder. If it has the file extension ".html" (in lower case, and using that exact form, i.e., not ".HTM" or some other variation) then any placeholders in the file will be expanded with their values. A few reserved placeholders have special effects, causing `cblorb` to expand interesting text in their places – see *Writing with Inform* for more on this.

`release source` ⟨*filename*⟩ using ⟨*filename*⟩ `from` ⟨*template*⟩

makes `cblorb` convert the Inform source text in the first filename into a suite of web pages using the style of the given file from the given template.

`website` ⟨*template*⟩

saves the best until last: it makes a complete website for an Inform project, using the named template. This means that the CSS file is copied into place (assuming `css` is used), the "index.html" is released from the template, the source of the project is run through `release source` using "source.html" from the template (assuming `source public` is used), and any extra files specified in the template's "(extras.txt)" are released as well. See *Writing with Inform* for more.

§**19.** An optional addition for a website is to incorporate a playable-in-browser form of the story, by base64-encoding the story file within a Javascript wrapper, then calling an interpreter such as Parchment.

The encoding part is taken care of by:

`base64` ⟨*filename*⟩ `to` ⟨*filename*⟩

This performs an RFC 1113-standard encoding on the binary file in (almost always our story file) into a textual base-64 file out. The file is topped and tailed with the text in placeholders `[BASESIXTYFOURTOP]` and `[BASESIXTYFOURTAIL]`, allowing Javascript wrapper code to surround the encoded data.

The interpreter itself is copied into place in the Release folder in a process rather like the construction of a website from a template. The necessary blurb command is:

`interpreter` ⟨*interpreter-name*⟩ ⟨*vm-letter*⟩

Interpreter names are like template names; Inform often uses "Parchment". The VM letter should be "g" if we need this to handle a Glulx story file (blorbed up), or "z" if we need it to handle a Z-machine story file. (This needs to be said because Inform doesn't have a way of knowing which formats a given interpreter can handle; so it has to leave checking to `cblorb` to do. Thus, if an Inform user tries to release a Z-machine-only interpreter with a Glulx story file, it's `cblorb` which issues the error, not Inform itself.)

§**20.** Finally (really finally this time), three commands to do with the "status" page, an HTML page written by `cblorb` to report back on what it has done. If requested, this is constructed for reading within the Inform application – it is not a valid HTML page in other contexts, and expects to have access to Javascript functions provided by Inform, and so on.

`status` ⟨*template*⟩ ⟨*filename*⟩
`status alternative` ⟨*link to Inform documentation*⟩
`status instruction` ⟨*link to Inform source text*⟩

The first simply requests the page to be made. It's made from a single template file, but in exactly the same way that website pages are generated from website templates – that is, placeholders are expanded. The second filename is where to write the result.

The other two commands allow Inform to insert information which `cblorb` otherwise has no access to: options for fancy release tricks not currently being used (with links to the documentation on them), and links to source text "Release along with..." sentences.