3 Other Material

Releaser 3/rel

Purpose

To manage requests to release material other than a Blorb file.

Definitions

¶1. If the previous section, "Blorb Writer.w", was the Lord High Executioner, then this one is the Lord High Everything Else: it keeps track of requests to write all kinds of interesting things which are *not* blorb files, and then sees that they are carried out. The requests divide as follows:

```
define COPY_REQ 0                        a miscellaneous file
define IFICTION_REQ 1                    the iFiction record of a project
define RELEASE_FILE_REQ 2                a template file
define RELEASE_SOURCE_REQ 3              the source text in HTML form
define SOLUTION_REQ 4                    a solution file generated from the skein
define SOURCE_REQ 5                      the source text of a project
define WEBSITE_REQ 6                     a whole website
define INTERPRETER_REQ 7                 an in-browser interpreter
define BASE64_REQ 8                      a base64-encoded copy of a binary file
define INSTRUCTION_REQ 9                 a release instruction copied to cblorb for reporting only
define ALTERNATIVE_REQ 10                an unused release instruction copied to cblorb for reporting only
```

```
int website_requested = FALSE;           has a WEBSITE_REQ been made?
```

¶2. This would use a lot of memory if there were many requests, but there are not and it does not.

```
typedef struct request {
    int what_is_requested;                    one of the *_REQ values above
    char details1[MAX_FILENAME_LENGTH];
    char details2[MAX_FILENAME_LENGTH];
    char details3[MAX_FILENAME_LENGTH];
    int private;                              is this request private, i.e., not to contribute to a website?
    int outcome_data;                         e.g. number of bytes copied
    MEMORY_MANAGEMENT
} request;
```

The structure request is private to this section.

§1. Receiving requests. These can have from 0 to 3 textual details attached:

```
request *request_0(int kind, int privacy) {
    request *req = CREATE(request);
    req->what_is_requested = kind;
    req->details1[0] = 0;
    req->details2[0] = 0;
    req->details3[0] = 0;
    req->private = privacy;
    req->outcome_data = 0;
    if (kind == WEBSITE_REQ) website_requested = TRUE;
    return req;
}
request *request_1(int kind, char *text1, int privacy) {
    request *req = request_0(kind, privacy);
    strcpy(req->details1, text1);
    return req;
}
request *request_2(int kind, char *text1, char *text2, int privacy) {
    request *req = request_0(kind, privacy);
    strcpy(req->details1, text1);
    strcpy(req->details2, text2);
    return req;
}
request *request_3(int kind, char *text1, char *text2, char *text3, int privacy) {
    request *req = request_0(kind, privacy);
    strcpy(req->details1, text1);
    strcpy(req->details2, text2);
    strcpy(req->details3, text3);
    return req;
}
```

The function request_0 is.
The function request_1 is.
The function request_2 is.
The function request_3 is.

§2. A convenient abbreviation:

```
void request_copy(char *from, char *to) {
    request_2(COPY_REQ, from, to, FALSE);
}
```

The function request_copy is called from 3/links.

§3. Any Last Requests. Most of the requests are made as the parser reads commands from the blurb script. At the end of that process, though, the following routine may add further requests as consequences:

```
void any_last_requests(void) {
    request_copy_of_auxiliaries();
    if (default_cover_used == FALSE) {
        char *BIGCOVER = read_placeholder("BIGCOVER");
        if (BIGCOVER) {
            if (cover_is_in_JPEG_format) request_copy(BIGCOVER, "Cover.jpg");
            else request_copy(BIGCOVER, "Cover.png");
        }
        if (website_requested) {
            char *SMALLCOVER = read_placeholder("SMALLCOVER");
            if (SMALLCOVER) {
                if (cover_is_in_JPEG_format) request_copy(SMALLCOVER, "Small Cover.jpg");
                else request_copy(SMALLCOVER, "Small Cover.png");
            }
        }
    }
}
```

The function any_last_requests is.

§4. Carrying out requests.

```
void create_requested_material(void) {
    if (release_folder[0] == 0) return;
    printf("! Release folder: <%s>\n", release_folder);
    if (blorb_file_size > 0) declare_where_blorb_should_be_copied(release_folder);
    any_last_requests();
    request *req;
    LOOP_OVER(req, request) {
        switch (req->what_is_requested) {
            case ALTERNATIVE_REQ: break;
            case BASE64_REQ: ⟨Copy a base64-encoded file across 9⟩; break;
            case COPY_REQ: ⟨Copy a file into the release folder 8⟩; break;
            case IFICTION_REQ: ⟨Create an iFiction file 7⟩; break;
            case INSTRUCTION_REQ: break;
            case INTERPRETER_REQ: ⟨Create an in-browser interpreter 12⟩; break;
            case RELEASE_FILE_REQ: ⟨Release a file into the release folder 10⟩; break;
            case RELEASE_SOURCE_REQ: ⟨Release source text as HTML into the release folder 11⟩; break;
            case SOLUTION_REQ: ⟨Create a walkthrough file 5⟩; break;
            case SOURCE_REQ: ⟨Create a plain text source file 6⟩; break;
            case WEBSITE_REQ: ⟨Create a website 13⟩; break;
        }
    }
}
```

The function create_requested_material is called from 1/main.

§5.

⟨Create a walkthrough file 5⟩ ≡
```
char Skein_filename[MAX_FILENAME_LENGTH];
sprintf(Skein_filename, "%s%cSkein.skein", project_folder, SEP_CHAR);
char solution_filename[MAX_FILENAME_LENGTH];
sprintf(solution_filename, "%s%csolution.txt", release_folder, SEP_CHAR);
walkthrough(Skein_filename, solution_filename);
```

This code is used in §4.

§6.

⟨Create a plain text source file 6⟩ ≡
```
char source_text_filename[MAX_FILENAME_LENGTH];
sprintf(source_text_filename, "%s%cSource%cstory.ni",
    project_folder, SEP_CHAR, SEP_CHAR);
char write_to[MAX_FILENAME_LENGTH];
sprintf(write_to, "%s%csource.txt", release_folder, SEP_CHAR);
copy_file(source_text_filename, write_to, FALSE);
```

This code is used in §4.

§7.

⟨Create an iFiction file 7⟩ ≡
```
char iFiction_filename[MAX_FILENAME_LENGTH];
sprintf(iFiction_filename, "%s%cMetadata.iFiction", project_folder, SEP_CHAR);
char write_to[MAX_FILENAME_LENGTH];
sprintf(write_to, "%s%ciFiction.xml", release_folder, SEP_CHAR);
copy_file(iFiction_filename, write_to, FALSE);
```

This code is used in §4.

§8.

⟨Copy a file into the release folder 8⟩ ≡
```
char write_to[MAX_FILENAME_LENGTH];
sprintf(write_to, "%s%c%s", release_folder, SEP_CHAR, req->details2);
int size = copy_file(req->details1, write_to, TRUE);
req->outcome_data = size;
if (size == -1) {
    int i;
    for (i = strlen(req->details1); i>0; i--)
        if ((req->details1)[i] == SEP_CHAR) { i++; break; }
    errorf_1s(
        "You asked to release along with a file called '%s', which ought "
        "to be in the Materials folder for the project. But I can't find "
        "it there.", (req->details1)+i);
}
```

This code is used in §4.

§9.

⟨Copy a base64-encoded file across 9⟩ ≡

```
encode_as_base64(req->details1, req->details2,
    read_placeholder("BASESIXTYFOURTOP"), read_placeholder("BASESIXTYFOURTAIL"));
```

This code is used in §4.

§10.

⟨Release a file into the release folder 10⟩ ≡

```
release_file_into_website(req->details1, req->details2, NULL);
```

This code is used in §4.

§11.

⟨Release source text as HTML into the release folder 11⟩ ≡

```
set_placeholder_to("SOURCEPREFIX", "source", 0);
set_placeholder_to("SOURCELOCATION", req->details1, 0);
set_placeholder_to("TEMPLATE", req->details3, 0);
char *HTML_template = find_file_in_named_template(req->details3, req->details2);
if (HTML_template == NULL) error_1("can't find HTML template file", req->details2);
if (trace_mode) printf("! Web page %s from template %s\n", HTML_template, req->details3);
web_copy_source(HTML_template, release_folder);
```

This code is used in §4.

§12. Interpreters are copied, not made. They're really just like website templates, except that they have a manifest file instead of an extras file, and that they're copied into an `interpreter` subfolder of the release folder, which is assumed already to exist. (It isn't copied because folder creation is tiresome to do in a cross-platform way, since Windows doesn't follow POSIX. The necessary code exists in Inform already, so we'll do it there.)

⟨Create an in-browser interpreter 12⟩ ≡

```
set_placeholder_to("INTERPRETER", req->details1, 0);
char *t = read_placeholder("INTERPRETER");
char *from = find_file_in_named_template(t, "(manifest).txt");
if (from) {                                  i.e., if the "(manifest).txt" file exists
    file_read(from, "can't open (manifest) file", FALSE, read_requested_ifile, 0);
}
```

This code is used in §4.

§13. We copy the CSS file, if we need one; make the home page; and make any other pages demanded by public released material. After that, it's up to the template to add more if it wants to.

```
⟨Create a website 13⟩ ≡
    set_placeholder_to("TEMPLATE", req->details1, 0);
    char *t = read_placeholder("TEMPLATE");
    if (use_css_code_styles) {
        char *from = find_file_in_named_template(t, "style.css");
        if (from) {
            char CSS_filename[MAX_FILENAME_LENGTH];
            sprintf(CSS_filename, "%s%cstyle.css", release_folder, SEP_CHAR);
            copy_file(from, CSS_filename, FALSE);
        }
    }
    release_file_into_website("index.html", t, NULL);
    request *req;
    LOOP_OVER(req, request)
        if (req->private == FALSE)
            switch (req->what_is_requested) {
                case INTERPRETER_REQ:
                    release_file_into_website("play.html", t, NULL); break;
                case SOURCE_REQ:
                    set_placeholder_to("SOURCEPREFIX", "source", 0);
                        char source_text[MAX_FILENAME_LENGTH];
                    sprintf(source_text, "%s%cSource%cstory.ni",
                        project_folder, SEP_CHAR, SEP_CHAR);
                    set_placeholder_to("SOURCELOCATION", source_text, 0);
                    release_file_into_website("source.html", t, NULL); break;
            }
    ⟨Add further material as requested by the template 14⟩;
```

This code is used in §4.

§14. Most templates do not request extra files, but they have the option by including a manifest called "(extras).txt":

```
⟨Add further material as requested by the template 14⟩ ≡
    char *from = find_file_in_named_template(t, "(extras).txt");
    if (from) {                                   i.e., if the "(extras).txt" file exists
        file_read(from, "can't open (extras) file", FALSE, read_requested_file, 0);
    }
```

This code is used in §13.

§15. The Extras file for a website template. When parsing "(extras).txt", `read_requested_file` is called for each line. We trim white space and expect the result to be a filename of something within the template.

```
void read_requested_file(char *filename, text_file_position *tfp) {
    filename = trim_white_space(filename);
    if (filename[0] == 0) return;
    release_file_into_website(filename, read_placeholder("TEMPLATE"), NULL);
}
```

The function read_requested_file is.

§16. The Manifest file for an interpreter. When parsing "(manifest).txt", we do almost the same thing. Like a website template, an interpreter is stored in a single folder, and the manifest can list files which need to be copied into the Release in order to piece together a working copy of the interpreter.

However, this is more expressive than the "(extras).txt" file because it also has the ability to set placeholders in `cblorb`. We use this mechanism because it allows each interpreter to provide some metadata about its own identity and exactly how it wants to be interfaced with the website which `cblorb` will generate. This isn't the place to document what those metadata placeholders are and what they mean, since (except for a consistency check below) `cblorb` doesn't know anything about them – it's the Standard website template which they need to match up to. Anyway, the best way to get an idea of this is to read the manifest file for the default, Parchment, interpreter.

```
char current_placeholder[MAX_VAR_NAME_LENGTH];
int cp_written = FALSE;
void read_requested_ifile(char *manifestline, text_file_position *tfp) {
    if (cp_written == FALSE) { cp_written = TRUE; current_placeholder[0] = 0; }
    manifestline = trim_white_space(manifestline);
    if (manifestline[0] == '[') ⟨Go into or out of placeholder setting mode 17⟩;
    if (current_placeholder[0] == 0)
        ⟨We're outside placeholder mode, so it's a comment or a manifested filename 18⟩
    else
        ⟨We're inside placeholder mode, so it's content to be spooled into the named placeholder 19⟩;
}
```

The function read_requested_ifile is.

§17. Placeholders are set thus:

```
    [INTERPRETERVERSION]
    Parchment for Inform 7
    []
```

where the opening line names the placeholder, then one or more lines give the contents, and the box line ends the definition.

We're in the mode if `current_placeholder` is a non-empty C string, and if so, then it's the name of the one being set. Thus the code to handle the opening and closing lines can be identical.

```
⟨Go into or out of placeholder setting mode 17⟩ ≡
    if (manifestline[strlen(manifestline)-1] == ']') {
        if (strlen(manifestline) >= MAX_VAR_NAME_LENGTH) {
            error_1("placeholder name too long in manifest file", manifestline);
            return;
        }
        strcpy(current_placeholder, manifestline+1);
        current_placeholder[strlen(current_placeholder)-1] = 0;
        return;
    }
    error_1("placeholder name lacks ']' in manifest file", manifestline);
    return;
```

This code is used in §16.

§18. Outside of placeholders, blank lines and lines introduced by the comment character `!` are skipped.

⟨We're outside placeholder mode, so it's a comment or a manifested filename 18⟩ ≡

```
    if ((manifestline[0] == '!') || (manifestline[0] == 0)) return;
    release_file_into_website(manifestline, read_placeholder("INTERPRETER"), "interpreter");
```

This code is used in §16.

§19. Line breaks are included between lines, though not at the end of the final line, so that a one-line definition like the example above contains no line break. White space is stripped out at the left and right hand edges of each line.

⟨We're inside placeholder mode, so it's content to be spooled into the named placeholder 19⟩ ≡

```
    if (strcmp(current_placeholder, "INTERPRETERVM") == 0)
        ⟨Check the value being given against the actual VM we're blorbing up 20⟩;
    if (read_placeholder(current_placeholder))
        append_to_placeholder(current_placeholder, "\n");
    append_to_placeholder(current_placeholder, manifestline);
```

This code is used in §16.

§20. Perhaps it's clumsy to do it here, but at some point `cblorb` needs to make sure we aren't trying to release a Z-machine game along with a Glulx interpreter, or vice versa. The manifest file for the interpreter is required to declare which virtual machines it implements, by giving a value of the placeholder `INTERPRETERVM`. This declares whether the interpreter can handle blorbed Z-machine files (`z`), blorbed Glulx files (`g`) or both (`zg` or `gz`). No other values are legal; note lower case. `cblorb` then checks this against its own placeholder `INTERPRETERVMIS`, which stores what the actual format of the blorb being released is.

⟨Check the value being given against the actual VM we're blorbing up 20⟩ ≡

```
    char *vm_used = read_placeholder("INTERPRETERVMIS");
    int i, capable = FALSE;
    for (i=0; manifestline[i]; i++)
        if (vm_used[0] == manifestline[i]) capable = TRUE;
    if (capable == FALSE) {
        char *format = "Z-machine";
        if (vm_used[0] == 'g') format = "Glulx";
        errorf_2s(
            "You asked to release along with a copy of the '%s' in-browser "
            "interpreter, but this can't handle story files which use the "
            "%s story file format. (The format can be changed on Inform's "
            "Settings panel for a project.)",
            read_placeholder("INTERPRETER"), format);
    }
```

This code is used in §19.

§21. There are really three cases when we release something from a website template. We can copy it verbatim as a binary file, we can expand placeholders but otherwise copy as a single item, or we can use it to make a mass generation of source pages.

```
void release_file_into_website(char *name, char *t, char *sub) {
    char write_to[MAX_FILENAME_LENGTH];
    if (sub) sprintf(write_to, "%s%c%s%c%s",
        release_folder, SEP_CHAR, sub, SEP_CHAR, name);
    else sprintf(write_to, "%s%c%s", release_folder, SEP_CHAR, name);

    char *from = find_file_in_named_template(t, name);
    if (from == NULL) {
        error_1("unable to find file in website template", name);
        return;
    }

    if (strcmp(get_filename_extension(name), ".html") == 0)
        ⟨Release an HTML page from the template into the website 22⟩
    else
        ⟨Release a binary file from the template into the website 23⟩;
}
```

The function release_file_into_website is.

§22. "Source.html" is a special case, as it expands into a whole suite of pages automagically. Otherwise we work out the filenames and then hand over to the experts.

```
⟨Release an HTML page from the template into the website 22⟩ ≡
    set_placeholder_to("TEMPLATE", t, 0);
    if (trace_mode) printf("! Web page %s from template %s\n", name, t);
    if (strcmp(name, "source.html") == 0)
        web_copy_source(from, release_folder);
    else
        web_copy(from, write_to);
```

This code is used in §21.

§23.

```
⟨Release a binary file from the template into the website 23⟩ ≡
    if (trace_mode) printf("! Binary file %s from template %s\n", name, t);
    copy_file(from, write_to, FALSE);
```

This code is used in §21.

§24. The home page will need links to any public released resources, and this is where those are added (to the other links already present, that is).

```
void add_links_to_requested_resources(FILE *COPYTO) {
    request *req;
    LOOP_OVER(req, request)
        if (req->private == FALSE)
            switch (req->what_is_requested) {
                case WEBSITE_REQ: break;
                case INTERPRETER_REQ:
                    fprintf(COPYTO, "<li>");
                    download_link(COPYTO, "Play In-Browser", NULL, "play.html", "link");
                    fprintf(COPYTO, "</li>");
                    break;
                case SOURCE_REQ:
                    fprintf(COPYTO, "<li>");
                    download_link(COPYTO, "Source Text", NULL, "source.html", "link");
                    fprintf(COPYTO, "</li>");
                    break;
                case SOLUTION_REQ:
                    fprintf(COPYTO, "<li>");
                    download_link(COPYTO, "Solution", NULL, "solution.txt", "link");
                    fprintf(COPYTO, "</li>");
                    break;
                case IFICTION_REQ:
                    fprintf(COPYTO, "<li>");
                    download_link(COPYTO, "Library Card", NULL, "iFiction.xml", "link");
                    fprintf(COPYTO, "</li>");
                    break;
            }
}
```

The function add_links_to_requested_resources is called from 3/links.

§25. Blorb relocation. This is a little dodge used to make the process of releasing games in Inform 7 more seamless: see the manual for an explanation.

```
void declare_where_blorb_should_be_copied(char *path) {
    char *leaf = read_placeholder("STORYFILE");
    if (leaf == NULL) leaf = "Story";
    printf("Copy blorb to: [[%s%c%s]]\n", path, SEP_CHAR, leaf);
}
```

The function declare_where_blorb_should_be_copied is.

§26. Reporting the release. Inform normally asks `cblorb` to generate an HTML page reporting what it has done, and if things have gone well then this typically contains a list of what has been released. (That's easy for us to produce, since we just have to look through the requests.) Rather than attempt to write to the file here, we copy the necessary HTML into the placeholder `ph`.

```
void report_requested_material(char *ph) {
    if (release_folder[0] == 0) return;                    this should never happen
    int launch_website = FALSE, launch_play = FALSE;
    append_to_placeholder(ph, "<ul>");
    ⟨Itemise the blorb file, possibly mentioning pictures and sounds 27⟩;
    ⟨Itemise the website, mentioning how many pages it has 28⟩;
    ⟨Itemise the interpreter 29⟩;
    ⟨Itemise the library card 30⟩;
    ⟨Itemise the solution file 31⟩;
    ⟨Itemise the source text 32⟩;
    ⟨Itemise auxiliary files in a sub-list 33⟩;
    append_to_placeholder(ph, "</ul>");
    if ((launch_website) || (launch_play))
        ⟨Give a centred line of links to the main web pages produced 34⟩;
    ⟨Add in links to release instructions from Inform source text 35⟩;
    ⟨Add in advertisements for features Inform would like to offer 36⟩;
}
```

The function report_requested_material is called from 1/main.

§27.

```
⟨Itemise the blorb file, possibly mentioning pictures and sounds 27⟩ ≡
    if ((no_pictures_included > 1) || (no_sounds_included > 0))
        append_to_placeholder(ph,
            "<li>The blorb file <b>[STORYFILE]</b> ([BLORBFILESIZE]K in size, "
            "including [BLORBFILEPICTURES] figures(s) and [BLORBFILESOUNDS] "
            "sound(s))</li>");
    else
        append_to_placeholder(ph,
            "<li>The blorb file <b>[STORYFILE]</b> ([BLORBFILESIZE]K in size)</li>");
```

This code is used in §26.

§28.

```
⟨Itemise the website, mentioning how many pages it has 28⟩ ≡
    if (count_requests_of_type(WEBSITE_REQ) > 0) {
        append_to_placeholder(ph,
            "<li>A website (generated from the [TEMPLATE] template) of ");
        char pcount[32];
        sprintf(pcount, "%d page%s", HTML_pages_created, (HTML_pages_created!=1)?"s":"");
        append_to_placeholder(ph, pcount);
        append_to_placeholder(ph, "</li>");
        launch_website = TRUE;
    }
```

This code is used in §26.

§29.

⟨Itemise the interpreter 29⟩ ≡

```
    if (count_requests_of_type(INTERPRETER_REQ) > 0) {
        launch_play = TRUE;
        append_to_placeholder(ph,
            "<li>A play-in-browser page (generated from the [INTERPRETER] interpreter)</li>");
    }
```

This code is used in §26.

§30.

⟨Itemise the library card 30⟩ ≡

```
    if (count_requests_of_type(IFICTION_REQ) > 0)
        append_to_placeholder(ph,
            "<li>The library card (stored as an iFiction record)</li>");
```

This code is used in §26.

§31.

⟨Itemise the solution file 31⟩ ≡

```
    if (count_requests_of_type(SOLUTION_REQ) > 0)
        append_to_placeholder(ph,
            "<li>A solution file</li>");
```

This code is used in §26.

§32.

⟨Itemise the source text 32⟩ ≡

```
    if (count_requests_of_type(SOURCE_REQ) > 0) {
        if (source_HTML_pages_created > 0) {
            append_to_placeholder(ph, "<li>The source text (as plain text and as ");
            char pcount[32];
            sprintf(pcount, "%d web page%s",
                source_HTML_pages_created, (source_HTML_pages_created!=1)?"s":"");
            append_to_placeholder(ph, pcount);
            append_to_placeholder(ph, ")</li>");
        }
    }
    if (count_requests_of_type(RELEASE_SOURCE_REQ) > 0)
        append_to_placeholder(ph,
            "<li>The source text (as part of the website)</li>");
```

This code is used in §26.

§**33.**

⟨Itemise auxiliary files in a sub-list 33⟩ ≡

```
    if (count_requests_of_type(COPY_REQ) > 0) {
        append_to_placeholder(ph, "<li>The following additional file(s):<ul>");
        request *req;
        LOOP_OVER(req, request)
            if (req->what_is_requested == COPY_REQ) {
                char *leafname = req->details2;
                append_to_placeholder(ph, "<li>");
                append_to_placeholder(ph, leafname);
                if (req->outcome_data >= 4096) {
                    char filesize[32];
                    sprintf(filesize, " (%dK)", req->outcome_data/1024);
                    append_to_placeholder(ph, filesize);
                } else if (req->outcome_data >= 0) {
                    char filesize[32];
                    sprintf(filesize, " (%d byte%s)",
                        req->outcome_data, (req->outcome_data!=1)?"s":"");
                    append_to_placeholder(ph, filesize);
                }
                append_to_placeholder(ph, "</li>");
            }
        append_to_placeholder(ph, "</ul></li>");
    }
```

This code is used in §26.

§**34.** These two links are handled by means of LAUNCH icons which, if clicked, open the relevant pages not in the Inform application but using an external web browser (e.g., Safari on most Mac OS X installations). We can only achieve this effect using a Javascript function provided by the Inform application, called `openUrl`.

⟨Give a centred line of links to the main web pages produced 34⟩ ≡

```
    append_to_placeholder(ph, "<p><center>");
    if (launch_website) {
        append_to_placeholder(ph,
            "<a href=\"[JAVASCRIPTPRELUDE]"
            "openUrl('file://[**MATERIALSFOLDERPATHOPEN]/Release/index.html')\">"
            "<img src='inform:/launch.png' border=0></a> home page");
    }
    if ((launch_website) && (launch_play))
        append_to_placeholder(ph, " : ");
    if (launch_play) {
        append_to_placeholder(ph,
            "<a href=\"[JAVASCRIPTPRELUDE]"
            "openUrl('file://[**MATERIALSFOLDERPATHOPEN]/Release/play.html')\">"
            "<img src='inform:/launch.png' border=0></a> play-in-browser page");
    }
    append_to_placeholder(ph, "</center></p>");
```

This code is used in §26.

§**35.** Since `cblorb` has no knowledge of what the Inform source text producing this blorb was, it can't finish the status report from its own knowledge – it must rely on details supplied to it by Inform via blurb commands. First, Inform gives it source-text links for any "Release along with..." sentences, which have by now become `INSTRUCTION_REQ` requests:

⟨Add in links to release instructions from Inform source text 35⟩ ≡

```
    request *req;
    int count = 0;
    LOOP_OVER(req, request)
        if (req->what_is_requested == INSTRUCTION_REQ) {
            if (count == 0)
                append_to_placeholder(ph, "<p>The source text gives release instructions ");
            else
                append_to_placeholder(ph, " and ");
            append_to_placeholder(ph, req->details1);
            append_to_placeholder(ph, " here");
            count++;
        }
    if (count > 0)
        append_to_placeholder(ph, ".</p>");
```

This code is used in §26.

§**36.** And secondly, Inform gives it adverts for other fancy services on offer, complete with links to the Inform documentation (which, again, `cblorb` doesn't itself know about); and these have by now become `ALTERNATIVE_REQ` requests.

⟨Add in advertisements for features Inform would like to offer 36⟩ ≡

```
    request *req;
    int count = 0;
    LOOP_OVER(req, request)
        if (req->what_is_requested == ALTERNATIVE_REQ) {
            if (count == 0)
                append_to_placeholder(ph,
                    "<p>Here are some other possibilities you might want to consider:<p><ul>");
            append_to_placeholder(ph, "<li>");
            append_to_placeholder(ph, req->details1);
            append_to_placeholder(ph, "</li>");
            count++;
        }
    if (count > 0)
        append_to_placeholder(ph, "</ul></p>");
```

This code is used in §26.

§**37.** A convenient way to see if we've received requests of any given type:

```
int count_requests_of_type(int t) {
    request *req;
    int count = 0;
    LOOP_OVER(req, request)
        if (req->what_is_requested == t)
            count++;
    return count;
}
```

The function count_requests_of_type is.