CBLORB

The Program

Chapter 2

Build 3/100422   Graham Nelson

`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 Blorbs

`2/blorb`: *Blorb Writer.w* To write the Blorb file, our main output, to disc.

Blorb Writer

Purpose

To write the Blorb file, our main output, to disc.

Definitions

¶1. "Blorb" is an IF-specific format, but it is defined as a form of IFF file. IFF, "Interchange File Format", is a general-purpose wrapper format dating back to the mid-1980s; it was designed as a way to gather together audiovisual media for use on home computers. (Though Electronic Arts among others used IFF files to wrap up entertainment material, Infocom, the pioneer of IF at the time, did not.) Each IFF file consists of a chunk, but any chunk can contain other chunks in turn. Chunks are identified with initial ID texts four characters long. In different domains of computing, people use different chunks, and this makes different sorts of IFF file look like different file formats to the end user. So we have TIFF for images, AIFF for uncompressed audio, AVI for movies, GIF for bitmap graphics, and so on.

¶2. Main variables:

```
int total_size_of_Blorb_chunks = 0;             ditto, but not counting the FORM header or the RIdx chunk
int no_indexed_chunks = 0;
```

¶3. As we shall see, chunks can be used for everything from a few words of copyright text to 100MB of uncompressed choral music.

Our IFF file will consist of a front part and then the chunks, one after another, in order of their creation. Every chunk has a type, a 4-character ID like `"AUTH"` or `"JPEG"`, specifying what kind of data it holds; some chunks are also given resource", " numbers which allow the story file to refer to them as it runs – the pictures, sound effects and the story file itself all have unique resource numbers. (These are called "indexed", because references to them appear in a special `RIdx` record in the front part of the file – the "resource index".)

```
typedef struct chunk_metadata {
    char filename[MAX_FILENAME_LENGTH];                          if the content is stored on disc
    char data_in_memory[MAX_FILENAME_LENGTH];                  if the content is stored in memory
    int length_of_data_in_memory;                      in bytes; or −1 if the content is stored on disc
    char *chunk_type;                                                pointer to a 4-character string
    char *index_entry;                                                                        ditto
    int resource_id;                                 meaningful only if this is a chunk which is indexed
    int byte_offset;                  from the start of the chunks, which is not quite the start of the IFF file
    int size;                                                                              in bytes
    MEMORY_MANAGEMENT
} chunk_metadata;
```

The structure chunk_metadata is private to this section.

§1. Big-endian integers. IFF files use big-endian integers, whereas `cBlorb` might or might not (depending on the platform it runs on), so we need routines to write 32, 16 or 8-bit values in explicitly big-endian form:

```
void four_word(FILE *F, int n) {
    fputc((n / 0x1000000)%0x100, F);
    fputc((n / 0x10000)%0x100, F);
    fputc((n / 0x100)%0x100, F);
    fputc((n)%0x100, F);
}

void two_word(FILE *F, int n) {
    fputc((n / 0x100)%0x100, F);
    fputc((n)%0x100, F);
}

void one_byte(FILE *F, int n) {
    fputc((n)%0x100, F);
}

void s_four_word(char *F, int n) {
    F[0] = (n / 0x1000000)%0x100;
    F[1] = (n / 0x10000)%0x100;
    F[2] = (n / 0x100)%0x100;
    F[3] = (n)%0x100;
}

void s_two_word(char *F, int n) {
    F[0] = (n / 0x100)%0x100;
    F[1] = (n)%0x100;
}

void s_one_byte(char *F, int n) {
    F[0] = (n)%0x100;
}
```

The function four_word is.
The function two_word is.
The function one_byte is.
The function s_four_word is.
The function s_two_word is.
The function s_one_byte is.

§2. Chunks. Although chunks can be written in a nested way – that's the whole point of IFF, in fact – we will always be writing a very flat structure, in which a single enclosing chunk (`FORM`) contains a sequence of chunks with no further chunks inside.

```
chunk_metadata *current_chunk = NULL;
```

§3. Each chunk is "added" in one of two ways. *Either* we supply a filename for an existing binary file on disc which will hold the data we want to write, *or* we supply a `NULL` filename and a `data` pointer to `length` bytes in memory.

```
void add_chunk_to_blorb(char *id, int resource_num, char *supplied_filename, char *index,
    char *data, int length) {
    if (chunk_type_is_legal(id) == FALSE)
        fatal("tried to complete non-Blorb chunk");
    if (index_entry_is_legal(index) == FALSE)
        fatal("tried to include mis-indexed chunk");
    current_chunk = CREATE(chunk_metadata);
    ⟨Set the filename for the new chunk 4⟩;
    current_chunk->chunk_type = id;
    current_chunk->index_entry = index;
    if (current_chunk->index_entry) no_indexed_chunks++;
    current_chunk->byte_offset = total_size_of_Blorb_chunks;
    current_chunk->resource_id = resource_num;
    ⟨Compute the size in bytes of the chunk 5⟩;
    ⟨Advance the total chunk size 6⟩;
    if (trace_mode)
        printf("! Begun chunk %s: fn is <%s> (innate size %d)\n",
            current_chunk->chunk_type, current_chunk->filename, current_chunk->size);
}
```

The function add_chunk_to_blorb is.

§4.

```
⟨Set the filename for the new chunk 4⟩ ≡
    if (data) {
        strcpy(current_chunk->filename, "(not from a file)");
        current_chunk->length_of_data_in_memory = length;
        int i;
        for (i=0; i<length; i++) current_chunk->data_in_memory[i] = data[i];
    } else {
        strcpy(current_chunk->filename, supplied_filename);
        current_chunk->length_of_data_in_memory = -1;
    }
```

This code is used in §3.

§5.

```
⟨Compute the size in bytes of the chunk 5⟩ ≡
    int size;
    if (data) {
        size = length;
    } else {
        size = (int) file_size(supplied_filename);
    }
    if (chunk_type_is_already_an_IFF(current_chunk->chunk_type) == FALSE)
        size += 8;                          allow 8 further bytes for the chunk header to be added later
    current_chunk->size = size;
```

This code is used in §3.

§6. Note the adjustment of `total_size_of_Blorb_chunks` so as to align the next chunk's position at a two-byte boundary – this betrays IFF's origin in the 16-bit world of the mid-1980s. Today's formats would likely align at four, eight or even sixteen-byte boundaries.

⟨Advance the total chunk size 6⟩ ≡

```
    total_size_of_Blorb_chunks += current_chunk->size;
    if ((current_chunk->size) % 2 == 1) total_size_of_Blorb_chunks++;
```

This code is used in §3.

§7. Our choice of chunks. We will generate only the following chunks with the above apparatus. The full Blorb specification does include others, but Inform doesn't need them.

The weasel words "with the above..." are because we will also generate two chunks separately: the compulsory `"FORM"` chunk enclosing the entire Blorb, and an indexing chunk, `"RIdx"`. Within this index, some chunks appear, but not others, and they are labelled with the "index entry" text.

```
char *legal_Blorb_chunk_types[] = {
    "AUTH", "(c) ", "Fspc", "RelN", "IFmd",        miscellaneous identifying data
    "JPEG", "PNG ",                                images in different formats
    "AIFF", "OGGV", "MIDI", "MOD ",                sound effects in different formats
    "ZCOD", "GLUL",                                story files in different formats
    NULL };

char *legal_Blorb_index_entries[] = {
    "Pict", "Snd ", "Exec", NULL };
```

§8. Because we are wisely paranoid:

```
int chunk_type_is_legal(char *type) {
    int i;
    if (type == NULL) return FALSE;
    for (i=0; legal_Blorb_chunk_types[i]; i++)
        if (strcmp(type, legal_Blorb_chunk_types[i]) == 0)
            return TRUE;
    return FALSE;
}

int index_entry_is_legal(char *entry) {
    int i;
    if (entry == NULL) return TRUE;
    for (i=0; legal_Blorb_index_entries[i]; i++)
        if (strcmp(entry, legal_Blorb_index_entries[i]) == 0)
            return TRUE;
    return FALSE;
}
```

The function chunk_type_is_legal is.

The function index_entry_is_legal is.

§9. Because it will make a difference to how we embed a file into our Blorb, we need to know whether the chunk in question is already an IFF in its own right. Only one type of chunk is, as it happens:

```
int chunk_type_is_already_an_IFF(char *type) {
    if (strcmp(type, "AIFF")==0) return TRUE;
    return FALSE;
}
```

The function chunk_type_is_already_an_IFF is.

§10. `"AUTH"`: author's name, as a null-terminated string.

```
void author_chunk(char *t) {
    if (trace_mode) printf("! Author: <%s>\n", t);
    add_chunk_to_blorb("AUTH", 0, NULL, NULL, t, strlen(t));
}
```

The function author_chunk is called from 1/blurb.

§11. `"(c) "`: copyright declaration.

```
void copyright_chunk(char *t) {
    if (trace_mode) printf("! Copyright declaration: <%s>\n", t);
    add_chunk_to_blorb("(c) ", 0, NULL, NULL, t, strlen(t));
}
```

The function copyright_chunk is called from 1/blurb.

§12. `"Fspc"`: frontispiece image ID number – which picture resource provides cover art, in other words.

```
void frontispiece_chunk(int pn) {
    if (trace_mode) printf("! Frontispiece is image %d\n", pn);
    char data[4];
    s_four_word(data, pn);
    add_chunk_to_blorb("Fspc", 0, NULL, NULL, data, 4);
}
```

The function frontispiece_chunk is called from 1/blurb.

§13. `"RelN"`: release number.

```
void release_chunk(int rn) {
    if (trace_mode) printf("! Release number is %d\n", rn);
    char data[2];
    s_two_word(data, rn);
    add_chunk_to_blorb("RelN", 0, NULL, NULL, data, 2);
}
```

The function release_chunk is called from 1/blurb.

§14. `"Pict"`: a picture, or image. This must be available as a binary file on disc, and in a format which Blorb allows: for Inform 7 use, this will always be PNG or JPEG. There can be any number of these chunks.

```
void picture_chunk(int n, char *fn) {
    char *p = get_filename_extension(fn);
    char *type = "PNG ";
    if (*p == '.') {
        p++;
        if ((*p == 'j') || (*p == 'J')) type = "JPEG";
    }
    add_chunk_to_blorb(type, n, fn, "Pict", NULL, 0);
    no_pictures_included++;
}
```

The function picture_chunk is called from 1/blurb.

§15. `"Snd "`: a sound effect. This must be available as a binary file on disc, and in a format which Blorb allows: for Inform 7 use, this is officially Ogg Vorbis or AIFF at present, but there has been repeated discussion about adding MOD ("SoundTracker") or MIDI files, so both are supported here.

There can be any number of these chunks, too.

```
void sound_chunk(int n, char *fn) {
    char *p = get_filename_extension(fn);
    char *type = "AIFF";
    if (*p == '.') {
        p++;
        if ((*p == 'o') || (*p == 'O')) type = "OGGV";
        else if ((*p == 'm') || (*p == 'M')) {
            if ((p[1] == 'i') || (p[1] == 'I')) type = "MIDI";
            else type = "MOD ";
        }
    }
    add_chunk_to_blorb(type, n, fn, "Snd ", NULL, 0);
    no_sounds_included++;
}
```

The function sound_chunk is called from 1/blurb.

§16. `"Exec"`: the executable program, which will normally be a Z-machine or Glulx story file. It's legal to make a blorb with no story file in, but Inform 7 never does this.

```
void executable_chunk(char *fn) {
    char *p = get_filename_extension(fn);
    char *type = "ZCOD";
    if (*p == '.') {
        if (p[strlen(p)-1] == 'x') type = "GLUL";
    }
    add_chunk_to_blorb(type, 0, fn, "Exec", NULL, 0);
}
```

The function executable_chunk is called from 1/blurb.

§17. `"IFmd"`: the bibliographic data (or "metadata") about the work of IF being blorbed up, in the form of an iFiction record. (The format of which is set out in the *Treaty of Babel* agreement.)

```
void metadata_chunk(char *fn) {
    add_chunk_to_blorb("IFmd", 0, fn, NULL, NULL, 0);
}
```

The function metadata_chunk is called from 1/blurb.

§18. Main construction.

```
void write_blorb_file(char *out) {
    if (NUMBER_CREATED(chunk_metadata) == 0) return;
    FILE *IFF = fopen(out, "wb");
    if (IFF == NULL) fatal_fs("can't open blorb file for output", out);
    int RIdx_size, first_byte_after_index;
    ⟨Calculate the sizes of the whole file and the index chunk 19⟩;
    ⟨Write the initial FORM chunk of the IFF file, and then the index 20⟩;
    if (trace_mode) ⟨Print out a copy of the chunk table 24⟩;
    chunk_metadata *chunk;
    LOOP_OVER(chunk, chunk_metadata) ⟨Write the chunk 21⟩;
    fclose(IFF);
}
```

The function write_blorb_file is called from 1/main.

§19. The bane of IFF file generation is that each chunk has to be marked up-front with an offset to skip past it. This means that, unlike with XML or other files having flexible-sized ingredients delimited by begin-end markers, we always have to know the length of a chunk before we start writing it.

That even extends to the file itself, which is a single IFF chunk of type `"FORM"`. So we need to think carefully. We will need the `FORM` header, then the header for the `RIdx` indexing chunk, then the body of that indexing chunk – with one record for each indexed chunk; and then room for all of the chunks we'll copy in, whether they are indexed or not.

```
⟨Calculate the sizes of the whole file and the index chunk 19⟩ ≡
    int FORM_header_size = 12;
    int RIdx_header_size = 12;
    int index_entry_size = 12;
    RIdx_size = RIdx_header_size + index_entry_size*no_indexed_chunks;
    first_byte_after_index = FORM_header_size + RIdx_size;
    blorb_file_size = first_byte_after_index + total_size_of_Blorb_chunks;
```

This code is used in §18.

§20. Each different IFF file format is supposed to provide its own "magic text" identifying what the file format is, and for Blorbs that text is "IFRS", short for "IF Resource".

⟨Write the initial FORM chunk of the IFF file, and then the index 20⟩ ≡

```
fprintf(IFF, "FORM");
four_word(IFF, blorb_file_size - 8);            offset to end of FORM after the 8 bytes so far
fprintf(IFF, "IFRS");                           magic text identifying the IFF as a Blorb

fprintf(IFF, "RIdx");
four_word(IFF, RIdx_size - 8);                  offset to end of RIdx after the 8 bytes so far
four_word(IFF, no_indexed_chunks);              i.e., number of entries in the index

chunk_metadata *chunk;
LOOP_OVER(chunk, chunk_metadata)
    if (chunk->index_entry) {
        fprintf(IFF, "%s", chunk->index_entry);
        four_word(IFF, chunk->resource_id);
        four_word(IFF, first_byte_after_index + chunk->byte_offset);
    }
```

This code is used in §18.

§21. Most of the chunks we put in exist on disc without their headers, but AIFF sound files are an exception, because those are IFF files in their own right; so they come with ready-made headers.

⟨Write the chunk 21⟩ ≡

```
int bytes_to_copy;
char *type = chunk->chunk_type;
if (chunk_type_is_already_an_IFF(type) == FALSE) {
    fprintf(IFF, "%s", type);
    four_word(IFF, chunk->size - 8);            offset to end of chunk after the 8 bytes so far
    bytes_to_copy = chunk->size - 8;            since here the chunk size included 8 extra
} else {
    bytes_to_copy = chunk->size;                whereas here the chunk size was genuinely the file size
}

if (chunk->length_of_data_in_memory >= 0)
    ⟨Copy that many bytes from memory 23⟩
else
    ⟨Copy that many bytes from the chunk file on the disc 22⟩;

if ((bytes_to_copy % 2) == 1) one_byte(IFF, 0);     as we allowed for above
```

This code is used in §18.

§**22.** Sometimes the chunk's contents are on disc:

⟨Copy that many bytes from the chunk file on the disc 22⟩ ≡

```
FILE *CHUNKSUB = fopen(chunk->filename, "rb");
if (CHUNKSUB == NULL) fatal_fs("unable to read data", chunk->filename);
else {
    int i;
    for (i=0; i<bytes_to_copy; i++) {
        int j = fgetc(CHUNKSUB);
        if (j == EOF) fatal_fs("chunk ran out incomplete", chunk->filename);
        one_byte(IFF, j);
    }
    fclose(CHUNKSUB);
}
```

This code is used in §21.

§**23.** And sometimes, for shorter things, they are in memory:

⟨Copy that many bytes from memory 23⟩ ≡

```
int i;
for (i=0; i<bytes_to_copy; i++) {
    int j = chunk->data_in_memory[i];
    one_byte(IFF, j);
}
```

This code is used in §21.

§**24.** For debugging purposes only:

⟨Print out a copy of the chunk table 24⟩ ≡

```
printf("! Chunk table:\n");
chunk_metadata *chunk;
LOOP_OVER(chunk, chunk_metadata)
    printf("! Chunk %s %06x %s %d <%s>\n",
        chunk->chunk_type, chunk->size,
        (chunk->index_entry)?(chunk->index_entry):"unindexed",
        chunk->resource_id,
        chunk->filename);
printf("! End of chunk table\n");
```

This code is used in §18.