FileIO Template B/iot

Purpose

Reading and writing external files, in the Glulx virtual machine only.

§**1. Language.** This whole template contains material used only if the "Glulx external files" element is part of Inform's current definition, so:

```
#IFDEF PLUGIN_FILES;
```

§**2. Structure.** The I7 kind of value "auxiliary-file" is an `-->` array, holding a memory structure containing information about external files. The following constants specify memory offsets and values. Note the safety value stored as the first word of the structure: this helps protect the routines below from accidents. (16339, besides being prime, is a number interesting to the author of Inform since it was the examination board identifying number of his school, and so had to be filled in on all of the many papers he sat during his formative years.)

```
Constant AUXF_MAGIC = 0; ! First word holds a safety constant
Constant AUXF_MAGIC_VALUE = 16339; ! Should be first word of any valid file structure
Constant AUXF_STATUS = 1; ! One of the following:
    Constant AUXF_STATUS_IS_CLOSED = 1; ! Currently closed, or perhaps doesn't exist
    Constant AUXF_STATUS_IS_OPEN_FOR_READ = 2;
    Constant AUXF_STATUS_IS_OPEN_FOR_WRITE = 3;
    Constant AUXF_STATUS_IS_OPEN_FOR_APPEND = 4;
Constant AUXF_BINARY = 2; ! False for text files (I7 default), true for binary
Constant AUXF_STREAM = 3; ! Stream for an open file (meaningless otherwise)
Constant AUXF_FILENAME = 4; ! Packed address of constant string
Constant AUXF_IFID_OF_OWNER = 5; ! UUID_ARRAY if owned by this project, or
    ! string array of IFID of owner wrapped in //...//, or NULL to leave open
```

§**3. Instances.** These structures are not dynamically created: they are precompiled by the NI compiler, already filled in with the necessary values. The following command generates them.

```
{-call:Plugins::Files::arrays}
```

§**4. Errors.** This is used for I/O errors of all kinds: it isn't within the Glulx-only code because one of the errors is to try to use these routines on the Z-machine.

```
[ FileIO_Error extf err_text  struc;
    if ((extf < 1) || (extf > NO_EXTERNAL_FILES)) {
        print "^*** Error on unknown file: ", (string) err_text, " ***^";
    } else {
        struc = TableOfExternalFiles-->extf;
        print "^*** Error on file '",
            (string) struc-->AUXF_FILENAME, "': ",
            (string) err_text, " ***^";
    }
    RunTimeProblem(RTP_FILEIOERROR);
    return 0;
];
```

§**5. Glulx Material.**

```
#IFDEF TARGET_GLULX;
```

§**6. Existence.** Determine whether a file exists on disc. Note that we have no concept of directories, or the file system structure on the host machine: indeed, it is entirely up to the Glulx VM what it does when asked to look for a file. By convention, though, files for a project are stored in the same folder as the story file when out in the wild; when a project is developed within the Inform user interface, they are either (for preference) stored in a `Files` subfolder of the `Materials` folder for a project, or else stored alongside the Inform project file.

```
[ FileIO_Exists extf  fref struc rv usage;
    if ((extf < 1) || (extf > NO_EXTERNAL_FILES)) rfalse;
    struc = TableOfExternalFiles-->extf;
    if ((struc == 0) || (struc-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE)) rfalse;
    if (struc-->AUXF_BINARY) usage = fileusage_BinaryMode;
    else usage = fileusage_TextMode;
    fref = glk_fileref_create_by_name(fileusage_Data + usage,
        Glulx_ChangeAnyToCString(struc-->AUXF_FILENAME), 0);
    rv = glk_fileref_does_file_exist(fref);
    glk_fileref_destroy(fref);
    return rv;
];
```

§**7. Readiness.** One of our problems is that a file might be being used by another application: perhaps even by another story file running in a second incarnation of Glulx, like a parallel world of which we can know nothing. We actually want to allow for this sort of thing, because one use for external files in I7 is as a sort of communications conduit for assisting applications.

Most operating systems solve this problem by means of locking a file, or by creating a second lock-file, the existence of which indicates ownership of the original. We haven't got much access to the file-system, though: what we do is to set the first character of the file to an asterisk to mark it as complete and ready for reading, or to a hyphen to mark it as a work in progress.

`FileIO_Ready` determines whether or not a file is ready to be read from: it has to exist on disc, and to be openable, and also to be ready in having this marker asterisk.

`FileIO_MarkReady` changes the readiness state of a file, writing the asterisk or hyphen into the initial character as needed.

```
[ FileIO_Ready extf  struc fref usage str ch;
    if ((extf < 1) || (extf > NO_EXTERNAL_FILES)) rfalse;
    struc = TableOfExternalFiles-->extf;
    if ((struc == 0) || (struc-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE)) rfalse;
    if (struc-->AUXF_BINARY) usage = fileusage_BinaryMode;
    else usage = fileusage_TextMode;
    fref = glk_fileref_create_by_name(fileusage_Data + usage,
        Glulx_ChangeAnyToCString(struc-->AUXF_FILENAME), 0);
    if (glk_fileref_does_file_exist(fref) == false) {
        glk_fileref_destroy(fref);
        rfalse;
    }
    str = glk_stream_open_file(fref, filemode_Read, 0);
    ch = glk_get_char_stream(str);
    glk_stream_close(str, 0);
    glk_fileref_destroy(fref);
    if (ch ~= '*') rfalse;
    rtrue;
];
[ FileIO_MarkReady extf readiness  struc fref str ch usage;
    if ((extf < 1) || (extf > NO_EXTERNAL_FILES))
        return FileIO_Error(extf, "tried to open a non-file");
    struc = TableOfExternalFiles-->extf;
    if ((struc == 0) || (struc-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE)) rfalse;
    if (struc-->AUXF_BINARY) usage = fileusage_BinaryMode;
    else usage = fileusage_TextMode;
    fref = glk_fileref_create_by_name(fileusage_Data + usage,
        Glulx_ChangeAnyToCString(struc-->AUXF_FILENAME), 0);
    if (glk_fileref_does_file_exist(fref) == false) {
        glk_fileref_destroy(fref);
        return FileIO_Error(extf, "only existing files can be marked");
    }
    if (struc-->AUXF_STATUS ~= AUXF_STATUS_IS_CLOSED) {
        glk_fileref_destroy(fref);
        return FileIO_Error(extf, "only closed files can be marked");
    }
    str = glk_stream_open_file(fref, filemode_ReadWrite, 0);
    glk_stream_set_position(str, 0, 0); ! seek start
    if (readiness) ch = '*'; else ch = '-';
    glk_put_char_stream(str, ch); ! mark as complete
    glk_stream_close(str, 0);
    glk_fileref_destroy(fref);
];
```

§8. Open File.

```
[ FileIO_Open extf write_flag append_flag
    struc fref str mode ix ch not_this_ifid owner force_header usage;
    if ((extf < 1) || (extf > NO_EXTERNAL_FILES))
        return FileIO_Error(extf, "tried to open a non-file");
    struc = TableOfExternalFiles-->extf;
    if ((struc == 0) || (struc-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE)) rfalse;
    if (struc-->AUXF_STATUS ~= AUXF_STATUS_IS_CLOSED)
        return FileIO_Error(extf, "tried to open a file already open");
    if (struc-->AUXF_BINARY) usage = fileusage_BinaryMode;
    else usage = fileusage_TextMode;
    fref = glk_fileref_create_by_name(fileusage_Data + usage,
        Glulx_ChangeAnyToCString(struc-->AUXF_FILENAME), 0);
    if (write_flag) {
        if (append_flag) {
            mode = filemode_WriteAppend;
            if (glk_fileref_does_file_exist(fref) == false)
                force_header = true;
        }
        else mode = filemode_Write;
    } else {
        mode = filemode_Read;
        if (glk_fileref_does_file_exist(fref) == false) {
            glk_fileref_destroy(fref);
            return FileIO_Error(extf, "tried to open a file which does not exist");
        }
    }
    str = glk_stream_open_file(fref, mode, 0);
    glk_fileref_destroy(fref);
    if (str == 0) return FileIO_Error(extf, "tried to open a file but failed");
    struc-->AUXF_STREAM = str;
    if (write_flag) {
        if (append_flag)
            struc-->AUXF_STATUS = AUXF_STATUS_IS_OPEN_FOR_APPEND;
        else
            struc-->AUXF_STATUS = AUXF_STATUS_IS_OPEN_FOR_WRITE;
        glk_stream_set_current(str);
        if ((append_flag == FALSE) || (force_header)) {
            print "- ";
            for (ix=6: ix <= UUID_ARRAY->0: ix++) print (char) UUID_ARRAY->ix;
            print " ", (string) struc-->AUXF_FILENAME, "^";
        }
    } else {
        struc-->AUXF_STATUS = AUXF_STATUS_IS_OPEN_FOR_READ;
        ch = FileIO_GetC(extf);
        if (ch ~= '-' or '*') { jump BadFile; }
        if (ch == '-')
```

```
            return FileIO_Error(extf, "tried to open a file which was incomplete");
        ch = FileIO_GetC(extf);
        if (ch ~= ' ') { jump BadFile; }
        ch = FileIO_GetC(extf);
        if (ch ~= '/') { jump BadFile; }
        ch = FileIO_GetC(extf);
        if (ch ~= '/') { jump BadFile; }
        owner = struc-->AUXF_IFID_OF_OWNER;
        ix = 3;
        if (owner == UUID_ARRAY) ix = 8;
        if (owner ~= NULL) {
            for (: ix <= owner->0: ix++) {
                ch = FileIO_GetC(extf);
                if (ch == -1) { jump BadFile; }
                if (ch ~= owner->ix) not_this_ifid = true;
                if (ch == ' ') break;
            }
            if (not_this_ifid == false) {
                ch = FileIO_GetC(extf);
                if (ch ~= ' ') { jump BadFile; }
            }
        }
        while (ch ~= -1) {
            ch = FileIO_GetC(extf);
            if (ch == 10 or 13) break;
        }
        if (not_this_ifid) {
            struc-->AUXF_STATUS = AUXF_STATUS_IS_CLOSED;
            glk_stream_close(str, 0);
            return FileIO_Error(extf,
                "tried to open a file owned by another project");
        }
    }
    return struc-->AUXF_STREAM;
    .BadFile;
    struc-->AUXF_STATUS = AUXF_STATUS_IS_CLOSED;
    glk_stream_close(str, 0);
    return FileIO_Error(extf, "tried to open a file which seems to be malformed");
];
```

§9. Close File.

```
[ FileIO_Close extf  struc;
    if ((extf < 1) || (extf > NO_EXTERNAL_FILES))
        return FileIO_Error(extf, "tried to open a non-file");
    struc = TableOfExternalFiles-->extf;
    if (struc-->AUXF_STATUS ~=
        AUXF_STATUS_IS_OPEN_FOR_READ or
        AUXF_STATUS_IS_OPEN_FOR_WRITE or
        AUXF_STATUS_IS_OPEN_FOR_APPEND)
        return FileIO_Error(extf, "tried to close a file which is not open");
    if ((struc-->AUXF_BINARY == false) &&
        (struc-->AUXF_STATUS ==
        AUXF_STATUS_IS_OPEN_FOR_WRITE or
        AUXF_STATUS_IS_OPEN_FOR_APPEND)) {
        glk_set_window(gg_mainwin);
    }
    if (struc-->AUXF_STATUS ==
        AUXF_STATUS_IS_OPEN_FOR_WRITE or
        AUXF_STATUS_IS_OPEN_FOR_APPEND) {
        glk_stream_set_position(struc-->AUXF_STREAM, 0, 0); ! seek start
        glk_put_char_stream(struc-->AUXF_STREAM, '*'); ! mark as complete
    }
    glk_stream_close(struc-->AUXF_STREAM, 0);
    struc-->AUXF_STATUS = AUXF_STATUS_IS_CLOSED;
];
```

§10. Get Character.

```
[ FileIO_GetC extf  struc;
    if ((extf < 1) || (extf > NO_EXTERNAL_FILES)) return -1;
    struc = TableOfExternalFiles-->extf;
    if (struc-->AUXF_STATUS ~= AUXF_STATUS_IS_OPEN_FOR_READ) return -1;
    return glk_get_char_stream(struc-->AUXF_STREAM);
];
```

§11. Put Character.

```
[ FileIO_PutC extf char  struc;
    if ((extf < 1) || (extf > NO_EXTERNAL_FILES)) return -1;
        return FileIO_Error(extf, "tried to write to a non-file");
    struc = TableOfExternalFiles-->extf;
    if (struc-->AUXF_STATUS ~=
        AUXF_STATUS_IS_OPEN_FOR_WRITE or
        AUXF_STATUS_IS_OPEN_FOR_APPEND)
        return FileIO_Error(extf,
            "tried to write to a file which is not open for writing");
    return glk_put_char_stream(struc-->AUXF_STREAM, char);
];
```

§**12. Print Line.** We read characters from the supplied file until the next newline character. (We allow for that to be encoded as either a single `0a` or a single `0d`.) Each character is printed, and at the end we print a newline.

```
[ FileIO_PrintLine extf ch  struc;
    if ((extf < 1) || (extf > NO_EXTERNAL_FILES))
        return FileIO_Error(extf, "tried to write to a non-file");
    struc = TableOfExternalFiles-->extf;
    for (::) {
        ch = FileIO_GetC(extf);
        if (ch == -1) rfalse;
        if (ch == 10 or 13) { print "^"; rtrue; }
        print (char) ch;
    }
];
```

§**13. Print Contents.** Repeating this until the file runs out is equivalent to the Unix command `cat`, that is, it copies the stream of characters from the file to the output stream. (This might well be another file, just as with `cat`, in which case we have a copy utility.)

```
[ FileIO_PrintContents extf tab  struc;
    if ((extf < 1) || (extf > NO_EXTERNAL_FILES))
        return FileIO_Error(extf, "tried to access a non-file");
    struc = TableOfExternalFiles-->extf;
    if (struc-->AUXF_BINARY)
        return FileIO_Error(extf, "printing text will not work with binary files");
    if (FileIO_Open(extf, false) == 0) rfalse;
    while (FileIO_PrintLine(extf)) ;
    FileIO_Close(extf);
    rtrue;
];
```

§**14. Print Text.** The following writes a given piece of text as the new content of the file, either as the whole file (if `append_flag` is false) or adding only to the end (if true).

```
[ FileIO_PutContents extf text append_flag  struc str ch;
    if ((extf < 1) || (extf > NO_EXTERNAL_FILES))
        return FileIO_Error(extf, "tried to access a non-file");
    struc = TableOfExternalFiles-->extf;
    if (struc-->AUXF_BINARY)
        return FileIO_Error(extf, "writing text will not work with binary files");
    str = FileIO_Open(extf, true, append_flag);
    if (str == 0) rfalse;
    @push say__p; @push say__pc;
    ClearParagraphing();
    PrintText(text);
    FileIO_Close(extf);
    @pull say__pc; @pull say__p;
    rfalse;
];
```

§15. Serialising Tables. The most important data structures to "serialise" – that is, to convert from their binary representations in memory into text representations in an external file – are Tables. Here we only carry out the file-handling; the actual translations are in "Tables.i6t".

```
[ FileIO_PutTable extf tab rv  struc;
    if ((extf < 1) || (extf > NO_EXTERNAL_FILES))
        return FileIO_Error(extf, "tried to write table to a non-file");
    struc = TableOfExternalFiles-->extf;
    if (struc-->AUXF_BINARY)
        return FileIO_Error(extf, "writing a table will not work with binary files");
    if (FileIO_Open(extf, true) == 0) rfalse;
    rv = TablePrint(tab);
    FileIO_Close(extf);
    if (rv) return RunTimeProblem(RTP_TABLE_CANTSAVE, tab);
    rtrue;
];
[ FileIO_GetTable extf tab  struc;
    if ((extf < 1) || (extf > NO_EXTERNAL_FILES))
        return FileIO_Error(extf, "tried to read table from a non-file");
    struc = TableOfExternalFiles-->extf;
    if (struc-->AUXF_BINARY)
        return FileIO_Error(extf, "reading a table will not work with binary files");
    if (FileIO_Open(extf, false) == 0) rfalse;
    TableRead(tab, extf);
    FileIO_Close(extf);
    rtrue;
];
```

§16. Z-Machine Stubs. These routines do the minimum possible, but equally, they only generate a run-time problem when there is no alternative.

```
#IFNOT; ! TARGET_GLULX
[ FileIO_Exists extf; rfalse; ];
[ FileIO_Ready extf; rfalse; ];
[ FileIO_GetC extf; return -1; ];
[ FileIO_PutTable extf tab;
    return FileIO_Error(extf, "external files can only be used under Glulx");
];
[ FileIO_MarkReady extf status; FileIO_PutTable(extf); ];
[ FileIO_GetTable extf tab; FileIO_PutTable(extf); ];
[ FileIO_PrintContents extf; FileIO_PutTable(extf); ];
[ FileIO_PutContents extf; FileIO_PutTable(extf); ];
#ENDIF; ! TARGET_GLULX
```

§17. Back To Core.

```
#IFNOT; ! PLUGIN_FILES
[ FileIO_GetC extf; return -1; ];
#ENDIF; ! PLUGIN_FILES
```