What is RTF?

Rich Text Format (RTF) is a method of encoding formatted text and graphics for use within applications or for data and formatting transfer between applications[1]. In this Microsoft's document RTF document format is explained in detail and an example of RTF reader application is given at the end of the document with some source code.

RTF syntax is similar with HTML but it is not so simple or easy to be hand-coded. So, MSWord or WordPad will do just fine in generating *.rtf documents that can be viewed on different platforms using other RTF viewer application.

But, since there are too many RTF tags that can combined in different ways you can not be sure that the document on the destination viewer will look just the same as on the source. As an example, WordPad does not support all RTF tags MSWord does, so it may seem that document has some errors or that some things are missing. On the other hand, MSWord will open every RTF document created with WordPad.

There are many other freeware and shareware text editors that can be found all over the Internet for viewing or editing RTF documents. There are also developer tools which can generate RTF document within the code.

Here, a C-library for generating RTF documents from C/C++ custom applications will be presented and library interface will be explained.

rtflib v1.0

rtflib v1.0 is freeware C-library for generating RTF documents from the source code. The library methods can be invoked from just any place in the application and no additional dependencies are required to use library except supplied header files and compiled library module.

Current library version is 1.0 and there will be modified releases in the future, but library will stay free for un-commercial use. In this release, the basic methods for generating RTF documents will be explained. For now, only MSWord shows generated RTF documents as they are (with no missing parts or page errors)[2].

As mentioned, library was tested on Windows platforms running MSOffice97 to MSOffice2003 with identical final results in generated RTF documents.

[1] Microsoft Office Word 2003 Rich Text Format (RTF) Specification.
[2] There could also be some differences between different versions of MSWord considering image scaling but starting from MSWord97 to MSWord2003 no other bugs where noticed for now.

RTF syntax

Here is show how simple RTF document looks like:

```
{\rtf1\ansi\deff0{\fonttbl{\f0\froman Times New Roman;}}{\colortbl;\red0\green0\blue0;}
{\info{\author Author_Name}}\sectd\pard\plain\fs20\f0 Hello, World !!!\par}
```

When opened in WordPad or MSWord the result will be a text *"Hello, World !!!"* in the first section and in the first paragraph of the document. So, what are the main parts of the document?

Header part

At the beginning there is control word **\rtfN** which defines current RTF document version (most likely it will stay 1 for a while). Next, the default character set is set to be **\ansi** and a default font is the first font from the font table, that is **\deff0**.

Following these default definitions there is embedded document font table with control word **\fonttbl**. It defines font position in the font table (**\f0** for the first and so on), font family (like **\froman**) and font face name (it is **Times New Roman** in this case).

Font table is followed by document color table defined by **\colortbl** control word. Here are included RGB triplets which will be used later in the document.

Document part

It is consisted of optional information part (**\info** control word) and a document text part. In the info part information like: author name, document title, company and other data can be written.

Next, there is first section control word **\sectd** and first paragraph control word **\pard**. Control word **\plain** determine paragraph as plain text paragraph. Paragraph uses first font from the font table with **\f0** control word and it is written with font size of 10 points (control word **\fsN** defines font size in double points value).

The only thing that is going to be seen in RTF viewer is text **"Hello, World !!!"**.

So, it is clear now that RTF syntax is very similar to HTML syntax with RTF control words that replace HTML tags and attributes at the same time.

Using rtflib v1.0

To use rtflib v1.0 in custom applications the following header files must be added to the project:

- *globals.h* to use rtflib v1.0 global library variables
- *errors.h* to get rtflib v1.0 function error codes
- *structures.h* with base rtflib v1.0 type definitions
- *rtflib.h* with current rtflib v1.0 interface definition

Finally, a link must be added to compiled *rtflib.lib* library module and library interface is ready to be used in application.

Creating empty RTF document

In order to create empty RTF document an example is shown below.

```
void main()
{
  // RTF document font and color table definition
  char font_list[] = “Times New Roman;Arial;”;
  char color_list[] = “0;0;0;255;0;0;192;192;192”;

  // Create new RTF document
  rtf_open( “Sample.rtf”, font_list, color_list );

  // Close RTF document
  rtf_close();
}
```

Use *rtf_open()* library method to create and open new RTF file. If the file already exists it will be overwritten. No support is added to modify existing RTF documents. Params of this method are described below:

- **char*** *filename*, holds new RTF document file name
- **char*** *fonts*, holds RTF document font table
- **char*** *colors*, holds RTF document color table

It is show in the code example above how font and color table should be organized.

Call *rtf_close()* library method when finished working with the document to close created RTF document file.

Now, it is possible to open created RTF document with RTF viewer and see the result[3].

Adding paragraph text to RTF document

Between *rtf_open()* and *rtf_close()* calls following should be written to add some simple paragraph text to the document.

```
  // Write simple paragraph text
  rtf_start_paragraph( “Hello, World !!!”, false );
```

This method has following arguments:

[3] There will be no text since this code creates empty RTF document.

- **char*** *text*, holds paragraph text that will appear
- **bool** *newPar*, true if this is new paragraph

Now, if sample project is compiled with described modifications the result in the RTF viewer will be in page will single line text **"Hello, World !!!"**. In this manner it is possible to pass large amount of text to the function (could be read from *.txt file as an example) an this text will be presented as paragraph text. Repeating this function call will produce additional paragraph text.

If *newPar* is set to **true** than the text from the paragraph will appear in new line.

Formatting paragraph

To obtain pointer to paragraph formatting structure use following method:

```
// Get current paragraph formatting
RTF_PARAGRAPH_FORMAT* pf = rtf_get_paragraphformat();
```

RTF_PARAGRAPH_STRUCTURE has following definition:

```
struct RTF_PARAGRAPH_FORMAT
{
       int paragraphBreak;           // Sets paragraph break type
       bool newParagraph;            // New paragraph
       bool defaultParagraph;        // Default paragraph formatting
       int paragraphAligment;        // Sets paragraph aligment
       int firstLineIndent;          // Sets first line indent (the default is 0)
       int leftIndent;               // Sets paragraph left indent (the default is 0)
       int rightIndent;              // Sets paragraph right indent (the default is 0)
       int spaceBefore;              // Sets space before paragraph (the default is 0)
       int spaceAfter;               // Sets space after paragraph (the default is 0)
       int lineSpacing;              // Sets line spacing in paragraph
       char* paragraphText;          // Sets paragraph text
       bool tabbedText;              // Sets paragraph tabbed text
       bool tableText;               // Sets paragraph table text

       bool paragraphTabs;                  // Paragraph has tabs
       struct RTF_TABS_FORMAT TABS;         // Paragraf RTF_TAB_FORMAT structure

       bool paragraphNums;                  // Paragraph is numbered (bulleted)
       struct RTF_NUMS_FORMAT NUMS;         // Paragraph RTF_NUMS_FORMAT structure

       bool paragraphBorders;               // Paragraph has borders
       struct RTF_BORDERS_FORMAT BORDERS;   // Paragraph RTF_BORDERS_FORMAT structure

       bool paragraphShading;               // Paragraph has shading
       struct RTF_SHADING_FORMAT SHADING;   // Paragraph RTF_SHADING_FORMAT structure

       struct RTF_CHARACTER_FORMAT CHARACTER; // Paragraph RTF_CHARACTER_FORMAT structure
};
```

Since this is the largest structure defined to be used by the library it will be explained part by part. The first part is general and it has following members:

```
int paragraphBreak;           // Sets paragraph break type
bool newParagraph;            // New paragraph
bool defaultParagraph;        // Default paragraph formatting
int paragraphAligment;        // Sets paragraph aligment
int firstLineIndent;          // Sets first line indent (the default is 0)
int leftIndent;               // Sets paragraph left indent (the default is 0)
int rightIndent;              // Sets paragraph right indent (the default is 0)
int spaceBefore;              // Sets space before paragraph (the default is 0)
int spaceAfter;               // Sets space after paragraph (the default is 0)
int lineSpacing;              // Sets line spacing in paragraph
char* paragraphText;          // Sets paragraph text
bool tabbedText;              // Sets paragraph tabbed text
bool tableText;               // Sets paragraph table text
```

Use *paragraphBreak* member to specify break type that goes before paragraph text. There are following paragraph break type specified:

- RTF_PARAGRAPHBREAK_NONE, defines no break
- RTF_PARAGRAPHBREAK_PAGE, defines page break
- RTF_PARAGRAPHBREAK_COLUMN, defines column break
- RTF_PARAGRAPHBREAK_LINE, defines line break

Setting *paragraphBreak* member to any of these values will result in specified break type. This is very simple way to organize documents in paragraphs, sections and pages.

Member *newParagraph* is set by *newPar* argument of the *rtf_start_paragraph()* function, and similarly members *tabbedText* and *paragraphText*. Member *defaultParagraph* is always **true** unless this value is directly changed. If it is set to **true** that means that paragraph formatting is always default (it does not inherit formatting from previous paragraph).

To set paragraph aligment use *paragraphAligment* member which can be one of the following:

- RTF_PARAGRAPHALIGN_LEFT, defines left aligment
- RTF_PARAGRAPHALIGN_CENTER, defines center aligment
- RTF_PARAGRAPHALIGN_RIGHT, defines right aligment
- RTF_PARAGRAPHALIGN_JUSTIFY, defines justified aligment

If indentation is required then an integer value[4] for *firstLineIndent* member can be set to define offset from page left margin. Also members *leftIndent* and *rightIndent* are used to specify left and right paragraph offsets.

To specify offset from previous or next paragraph members *spaceBefore* and *spaceAfter* should be used.

For line offset member *lineSpacing* is used.

If paragraph is part of the table cell (will be discussed later) then member *tableText* should be set to **true**.

Now, as a conclusion, it is clear that these general paragraph formatting options can be used no matter if the paragraph is just plain text or if it is column text or even table text. RTF reader will obey to these formatting rules.

Next, some advanced paragraph formatting will be discussed. First, to write tabbed text there is an example below:

```
// Write tabbed paragraph text
pf->TABS.tabKind = RTF_PARAGRAPHTABKIND_RIGHT;
pf->TABS.tabLead = RTF_PARAGRAPHTABLEAD_DOT;
pf->TABS.tabPosition = 7200;
pf->paragraphTabs = true;
pf->tabbedText = false;
rtf_start_paragraph( "This is paragraph text", true );
pf->paragraphTabs = false;
pf->tabbedText = true;
rtf_start_paragraph( "This is tabbed text", false );
pf->paragraphTabs = false;
pf->tabbedText = false;
```

The structure that is necessary to be defined in this case is following:

[4] The most of the values that are passed as function arguments are considered to be in twips (1/20 of the point).

```
struct RTF_TABS_FORMAT
{
       int tabPosition;       // Sets tab position in twips from left margin
       int tabKind;           // Sets tab kind
       int tabLead;           // Sets tab lead
};
```

The member *TABS* of the paragraph formatting structure is of this type. *tabPosition* takes values which identifies offset from left page margin. *tabKind* member can be one of the following:

- RTF_PARAGRAPHTABKIND_NONE, defines no tab
- RTF_PARAGRAPHTABKIND_CENTER, defines center tab
- RTF_PARAGRAPHTABKIND_RIGHT, defines right tab
- RTF_PARAGRAPHTABKIND_DECIMAL, defines decimal tab

Similarly, *tabLead* member defines tab leader. It can one of the following:

- RTF_PARAGRAPHTABLEAD_NONE, defines no tab leader
- RTF_PARAGRAPHTABLEAD_DOT, defines dotted tab leader
- RTF_PARAGRAPHTABLEAD_MDOT, defines middle dots tab leader
- RTF_PARAGRAPHTABLEAD_HYPH, defines hyphens tab leader
- RTF_PARAGRAPHTABLEAD_UNDERLINE, defines underline tab leader
- RTF_PARAGRAPHTABLEAD_THICKLINE, defines thick line tab leader
- RTF_PARAGRAPHTABLEAD_EQUAL, defines equal sign tab leader

So, in above example the first paragraph line of text is a new paragraph text which is said to have tabs (*paragraphTabs* member is set to **true**). The second line of paragraph text will appear tab moved (7200 twips away) from the previous. Also, as there is defined tab leader type, these two paragraph texts will be connected by dots.

To write bulleted or numbered text see example:

```
// Write bulleted paragraph text
pf->paragraphNums = true;
pf->NUMS.numsLevel = 11;
pf->NUMS.numsSpace = 360;
pf->NUMS.numsChar = char(0x95);
rtf_start_paragraph( "Bulleted text1", true );
rtf_start_paragraph( "Bulleted text2", true );
rtf_start_paragraph( "Bulleted text3", true );
rtf_start_paragraph( "Bulleted text4", true );
rtf_start_paragraph( "Bulleted text5", true );
pf->paragraphNums = false;
```

The structure for formatting bullets and numbering is defined as follows:

```
struct RTF_NUMS_FORMAT
{
       int numsLevel;         // Sets paragraph numbered level (11 is bulleted paragraph)
       int numsSpace;         // Sets paragraph text distance from bulleted char
       char numsChar;         // Sets paragraph bullet char
};
```

The *NUMS* member of the paragraph formatting structure is of this type. *numsLevel* can receive values from 1 to 11. If the value is 11 than paragraph is considered to be bulleted and not numbered. *numsSpace* member requires previous member to have value of 11 (that is, paragraph should be declared as bulleted). It sets offset in twips from character which is used as bullet. Bullet character is set through *numsChar* member. It can take any ASCII value (from 0 to 255) to be used as bullet char. By default it is **0x95**. Paragraph is set to be bulleted or numbered through *paragraphNums* member (when set to **true**).

As a result, the page in RTF viewer will show five lines of bulleted text. Each line of paragraph bulleted or numbered text is declared as new paragraph in order to be shown in the new line.

Each paragraph can have borders around the text it contains, as in the following example:

```
// Draw border around paragraph text
pf->paragraphBorders = true;
pf->BORDERS.borderKind = RTF_PARAGRAPHBORDERKIND_BOX;
pf->BORDERS.borderType = RTF_PARAGRAPHBORDERTYPE_ENGRAVE;
pf->BORDERS.borderWidth = 0;
pf->BORDERS.borderColor = 0;
pf->BORDERS.borderSpace = 120;
rtf_start_paragraph( "This paragraph text has borders...", false );
pf->paragraphBorders = false;
```

The structure which describes paragraph border formatting is shown below:

```
struct RTF_BORDERS_FORMAT
{
       int borderKind;       // Sets paragraph border kind
       int borderType;       // Sets paragraph border type
       int borderWidth;      // Sets paragraph border width in twips
       int borderColor;      // Sets paragraph border color
       int borderSpace;      // Sets border distance from paragraph
};
```

The member *BORDERS* from paragraph formatting structure is of this type. *borderKind* member can be one of the following:

- RTF_PARAGRAPHBORDERKIND_NONE, defines no border
- RTF_PARAGRAPHBORDERKIND_TOP, defines top border
- RTF_PARAGRAPHBORDERKIND_BOTTOM, defines bottom border
- RTF_PARAGRAPHBORDERKIND_LEFT, defines left border
- RTF_PARAGRAPHBORDERKIND_RIGHT, defines right border
- RTF_PARAGRAPHBORDERKIND_BOX, defines box border

Member *borderType* can take following values:

- RTF_PARAGRAPHBORDERTYPE_STHICK, defines single thickness border
- RTF_PARAGRAPHBORDERTYPE_DTHICK, defines double thickness border
- RTF_PARAGRAPHBORDERTYPE_SHADOW, defines shadowed border
- RTF_PARAGRAPHBORDERTYPE_DOUBLE, defines double border
- RTF_PARAGRAPHBORDERTYPE_DOT, defines dotted border
- RTF_PARAGRAPHBORDERTYPE_DASH, defines dashed border
- RTF_PARAGRAPHBORDERTYPE_HAIRLINE, defines hairline border
- RTF_PARAGRAPHBORDERTYPE_INSET, defines inset border
- RTF_PARAGRAPHBORDERTYPE_SDASH, defines small dashed border
- RTF_PARAGRAPHBORDERTYPE_DOTDASH, defines dot-dashed border
- RTF_PARAGRAPHBORDERTYPE_DOTDOTDASH, defines dot-dot-dashed border
- RTF_PARAGRAPHBORDERTYPE_OUTSET, defines outset border
- RTF_PARAGRAPHBORDERTYPE_TRIPLE, defines triple border
- RTF_PARAGRAPHBORDERTYPE_WAVY, defines wavy border
- RTF_PARAGRAPHBORDERTYPE_DWAVY, defines double wavy border
- RTF_PARAGRAPHBORDERTYPE_STRIPED, defines striped border
- RTF_PARAGRAPHBORDERTYPE_EMBOSS, defines embosses border
- RTF_PARAGRAPHBORDERTYPE_ENGRAVE, defines engraved border

borderWidth member is used to set width of the border in twips (maximal width is 75). Also, color of the border can be set using *borderColor* member. The color is from the RTF document color table from the header part. Space between border and the enclosed paragraph text is set by *borderSpace* member.

In order to have shaded paragraph text see example code:

```
// Write shaded paragraph text
pf->paragraphShading = true;
pf->SHADING.shadingIntensity = 0;
pf->SHADING.shadingType = RTF_PARAGRAPHSHADINGTYPE_FILL;
pf->SHADING.shadingFillColor = 0;
pf->SHADING.shadingBkColor = 2;
rtf_start_paragraph( "This paragraph text is shaded...", false );
pf->paragraphShading = false;
```

The structure which defines shading is given below:

```
struct RTF_SHADING_FORMAT
{
       int shadingIntensity;  // Sets paragraph shading intensity
       int shadingType;       // Sets paragraph shading type
       int shadingFillColor;  // Sets paragraph shading fill color
       int shadingBkColor;    // Sets paragraph shading background color
};
```

The member *SHADING* from paragraph formatting structure corresponds to this structure. To set shading intensity *shadingIntensity* member should be used. *shadingType* member can have following values:

- RTF_PARAGRAPHSHADINGTYPE_FILL, defines fill shading
- RTF_PARAGRAPHSHADINGTYPE_HORIZ, defines horizontal shading
- RTF_PARAGRAPHSHADINGTYPE_VERT, defines vertical shading
- RTF_PARAGRAPHSHADINGTYPE_FDIAG, defines forward diagonal shading
- RTF_PARAGRAPHSHADINGTYPE_BDIAG, defines backward diagonal shading
- RTF_PARAGRAPHSHADINGTYPE_CROSS, defines cross shading
- RTF_PARAGRAPHSHADINGTYPE_CROSSD, defines cross diagonal shading
- RTF_PARAGRAPHSHADINGTYPE_DHORIZ, defines dark horizontal shading
- RTF_PARAGRAPHSHADINGTYPE_DVERT, defines dark vertical shading
- RTF_PARAGRAPHSHADINGTYPE_DFDIAG, defines dark forward diagonal shading
- RTF_PARAGRAPHSHADINGTYPE_DBDIAG, defines dark backward diagonal shading
- RTF_PARAGRAPHSHADINGTYPE_DCROSS, defines dark cross shading
- RTF_PARAGRAPHSHADINGTYPE_DCROSSD, defines dark cross diagonal shading

If fill shading is required then *shadingFillColor* will keep fill color and *shadingBkColor* will keep background color (both from document color table).

IMPORTANT NOTICE:

Please note that in all examples it is explicitly shown that it is necessary to control paragraph text flow using formatting switches members from paragraph formatting structure. When some formatting is needed these members are set to **true** and when specified formatting is no longer needed these members are again set to **false**. This means that *rtflib v1.0* behaves like some final state machine.

The final part considering paragraph text formatting is going to cover basic character formatting (that is font formatting). See example:

```
// Write bold underline text
pf->CHARACTER.boldCharacter = true;
pf->CHARACTER.underlineCharacter = 1;
rtf_start_paragraph( "This text is bold and underlined...", true );
pf->CHARACTER.boldCharacter = false;
pf->CHARACTER.underlineCharacter = 0;
```

The member *CHARACTER* from paragraph formatting structure is of the following type:

```
struct RTF_CHARACTER_FORMAT
{
       int animatedCharacter;        // Sets animated text
       bool boldCharacter;           // Sets text to bold
       bool capitalCharacter;        // Sets text to capital
       int backgroundColor;          // Sets text background color (the default is 0)
       int foregroundColor;          // Sets text foreground color (the default is 0)
       int scaleCharacter;           // Sets text scaling value (the default is 100)
       bool embossCharacter;         // Sets text to embossed
       int expandCharacter;          // Sets expansion or compression of the text
       int fontNumber;               // Sets font number
       int fontSize;                 // Sets font size (the default is 24)
       bool italicCharacter;         // Sets text to italic
       bool engraveCharacter;        // Sets text to engrave
       int kerningCharacter;         // Sets kerning of the text
       bool outlineCharacter;        // Sets text to outline
       bool smallcapitalCharacter;   // Sets text to small capital
       bool shadowCharacter;         // Sets text shadow
       bool strikeCharacter;         // Sets text to striketrough
       bool doublestrikeCharacter;   // Sets text to double striketrough
       bool subscriptCharacter;      // Sets text to subscript
       bool superscriptCharacter;    // Sets text to superscript
       int underlineCharacter;       // Sets text to underline

};
```

Use members from this structure to give effects to fonts used for the paragraph text.

Formatting section

Paragraph belongs to section. The section is a series of paragraphs. To get pointer to section formatting structure use following:

```
// Get current section formatting
RTF_SECTION_FORMAT* sf = rtf_get_sectionformat();
```

RTF_SECTION_FORMAT has following definition:

```
struct RTF_SECTION_FORMAT
{
       int sectionBreak;             // Sets section break type
       bool newSection;              // New section
       bool defaultSection;          // Default section formatting
       int pageWidth;                // Sets new page width in twips
       int pageHeight;               // Sets new page height in twips
       int pageMarginLeft;           // Sets new page left margin in twips
       int pageMarginRight;          // Sets new page right margin in twips
       int pageMarginTop;            // Sets new page top margin in twips
       int pageMarginBottom;         // Sets new page bottom margin in twips
       int pageGutterWidth;          // Sets new page gutter width in twips
       int pageHeaderOffset;         // Sets page header offset in twips
       int pageFooterOffset;         // Sets page footer offset in twips
       bool showPageNumber;          // Show page number
       int pageNumberOffsetX;        // Sets page number right offset
       int pageNumberOffsetY;        // Sets page number top offset
       bool cols;                    // Sets column section format
       int colsNumber;               // Sets number of columns in section
       int colsDistance;             // Sets distance between columns in twips
       bool colsLineBetween;         // Sets line between columns
};
```

This structure will also be explained in parts. The first part is more general and it is about formatting document page. So, following members are here:

```
int pageWidth;               // Sets new page width in twips
int pageHeight;              // Sets new page height in twips
int pageMarginLeft;          // Sets new page left margin in twips
int pageMarginRight;         // Sets new page right margin in twips
int pageMarginTop;           // Sets new page top margin in twips
int pageMarginBottom;        // Sets new page bottom margin in twips
int pageGutterWidth;         // Sets new page gutter width in twips
int pageHeaderOffset;        // Sets page header offset in twips
int pageFooterOffset;        // Sets page footer offset in twips
bool showPageNumber;         // Show page number
int pageNumberOffsetX;       // Sets page number right offset
int pageNumberOffsetY;       // Sets page number top offset
```

pageWidth and *pageHeight* members of this structure can be used in order to set page dimensions (in twips). Also, to set margins of the page use *pageMarginLeft*, *pageMarginRight*, *pageMarginTop* and *pageMarginBottom* members. Gutter size is set through *pageGutterWidth* member. An offset to page header or footer is set through *pageHeaderOffset* and *pageFooterOffset* members.

If page number should appear on the page then *showPageNumber* member has to be set to **true** and page number offsets from right and top margin should be specified using *pageNumberOffsetX* and *pageNumberOffetY* members.

To break one section and start another *sectionBreak* member can take following values:

- RTF_SECTIONBREAK_CONTINUOUS, defines continuous section break
- RTF_SECTIONBREAK_COLUMN, defines column section break
- RTF_SECTIONBREAK_PAGE, defines page section break
- RTF_SECTIONBREAK_EVENPAGE, defines even-page section break
- RTF_SECTIONBREAK_ODDPAGE, defines odd-page section break

There is an example below showing two lines of paragraph text in different sections.

```
// Create first section
rtf_start_section();
// Write paragraph text
rtf_start_paragraph( "First section text...", false );
// Create second section
rtf_start_section();
// Write paragraph text
rtf_start_paragraph( "Section section text...", false );
```

By default, section break is set to be continuous.

Now, sections are used to create column text. See example below:

```
sf->cols = true;
sf->colsDistance = 720;
sf->colsLineBetween = true;
sf->colsNumber = 2;
// Create new section
rtf_start_section();
// Write paragraph text
rtf_start_paragraph( "Column text is here...", false );
// Format paragraph
pf->paragraphBreak = RTF_PARAGRAPHBREAK_COLUMN;
// Write paragraph text
rtf_start_paragraph( "Column text also here...", false );
sf->cols = false;
```

Section formatting structure has also these members:

```
bool cols;                    // Sets column section format
int colsNumber;               // Sets number of columns in section
int colsDistance;             // Sets distance between columns in twips
bool colsLineBetween;         // Sets line between columns
```

Set number of columns using *colsNumber* member and column distance using *colsDistance* member. If there should be line between columns then *colsLineBetween* member should be set to **true**.

So, after declaring that section is column-formatted following paragraph text is added to first column. In order to break a column set *paragraphBreak* member to column break. Next paragraph is added to second column and so on.

Adding images to RTF document

Images can be easily inserted into the document. For this release, there are only few basic image types that are supported (*.bmp, *.jpg, *.gif) but they are often used. See example below:

```
// Load image (*.bmp, *.jpg, *.gif)
rtf_load_image("D:\\Slike\\Picture.jpg", 100, 100);
```

Library function *rtf_load_image()* has following arguments:

- **char*** *image*, holds image file name
- **int** *width*, holds image width
- **int** *height*, holds image height

Use *width* and *height* function arguments to pass scaling size for the image.

IMPORTANT NOTICE:

It is possible that image won’t be displayed at all in WordPad (bug that is noticed and hoped to be repaired in the next release).

Adding tables to RTF document

Here, the things are very simple. Tables are series of table rows. Table rows are series of table cells. Table cells are series of paragraphs. Any formatting that can be applied to plain text paragraph can also be applied to paragraph enclosed in table cell. Images can be inserted in the table cell and also bullets and numbering. Only tabs are not allowed inside table because behavior of the RTF viewer can not be defined in such case.

Table structure created with *rtflib v1.0* should look shown exactly the same in any RTF viewer, but table cells content display is not guaranteed, except on MSWord97 to MSWord2003.

To create simple table see code below:

```
// Format table row
RTF_TABLEROW_FORMAT* rf = rtf_get_tablerowformat();
rf->rowAligment = RTF_ROWTEXTALIGN_CENTER;
rf->marginTop = 120;
rf->marginBottom = 120;
rf->marginLeft = 120;
rf->marginRight = 120;
// Start table row
rtf_start_tablerow();
// Format table cell
RTF_TABLECELL_FORMAT* cf = rtf_get_tablecellformat();
cf->textVerticalAligment = RTF_CELLTEXTALIGN_CENTER;
cf->textDirection = RTF_CELLTEXDIRECTION_LRTB;
cf->borderBottom.border = true;
cf->borderBottom.BORDERS.borderType = RTF_PARAGRAPHBORDERTYPE_STHICK;
cf->borderBottom.BORDERS.borderWidth = 5;
cf->borderLeft.border = true;
cf->borderLeft.BORDERS.borderType = RTF_PARAGRAPHBORDERTYPE_STHICK;
cf->borderLeft.BORDERS.borderWidth = 5;
cf->borderRight.border = true;
cf->borderRight.BORDERS.borderType = RTF_PARAGRAPHBORDERTYPE_STHICK;
cf->borderRight.BORDERS.borderWidth = 30;
cf->borderTop.border = true;
cf->borderTop.BORDERS.borderType = RTF_PARAGRAPHBORDERTYPE_STHICK;
cf->borderTop.BORDERS.borderWidth = 5;
// Start table cell
rtf_start_tablecell(2000);
// Format table cell
cf->borderLeft.BORDERS.borderWidth = 30;
cf->borderRight.BORDERS.borderWidth = 5;
cf->cellShading = true;
cf->SHADING.shadingType = RTF_CELLSHADINGTYPE_FILL;
cf->SHADING.shadingBkColor = 2;
cf->SHADING.shadingIntensity = 20;
// Start table cell
rtf_start_tablecell(4000);
// Format paragraph
pf->tableText = true;
pf->paragraphAligment = RTF_PARAGRAPHALIGN_JUSTIFY;
// Write paragraph text
rtf_start_paragraph( "This is table cell text...", false );
rtf_start_paragraph( "These paragraphs are enclosed in table cell", true );
// End table cell
rtf_end_tablecell();
// Write paragraph text
rtf_start_paragraph( "This text is in another cell...", false );
rtf_start_paragraph( "Define correct cell right margin in order to see the text", true );
// End table cell
rtf_end_tablecell();
// End table row
rtf_end_tablerow();
pf->tableText = false;
```

A pointer to table row formatting structure should be obtained using *rtf_get_tablerowformat()* function. The structure is shown below:

```
struct RTF_TABLEROW_FORMAT
{
       int rowAligment;       // Sets table row aligment
       int rowHeight;         // Sets table row height (the default is 0)
       int marginLeft;        // Sets default cell left margin
       int marginRight;       // Sets default cell right margin
       int marginTop;         // Sets default cell top margin
       int marginBottom;      // Sets default cell bottom margin
       int rowLeftMargin;     // Sets default row left margin
};
```

Use *marginLeft*, *marginRight*, *marginTop* and *marginBottom* members to set margins of the table cell. Row height is specified using *rowHeight* member and row aligment using *rowAligment* member. For specifying left margin of the table row use *rowLeftMargin* member.

Row alignment member can take following values:

- RTF_ROWTEXALIGN_LEFT, defines left text alignment
- RTF_ROWTEXTALIGN_CENTER, defines center text alignment
- RTF_ROWTEXTALIGN_RIGHT, defines right text alignment

After having table row formatted call *rtf_start_tablerow()* function to write table row formatting to the RTF document. Each table row must be closed by calling *rtf_end_tablerow()* function before calling again this function. If it is not done in this manner the behavior of the RTF viewer can not be determined.

After formatting table row, but before adding any data to the table get pointer to table cell formatting structure using *rtf_get_tablecellformat()* function. The structure is shown below:

```
struct RTF_TABLECELL_FORMAT
{
       int textVerticalAligment;                    // Sets text vertical aligment
       int marginLeft;                              // Sets cell left margin
       int marginRight;                             // Sets cell right margin
       int marginTop;                               // Sets cell top margin
       int marginBottom;                            // Sets cell bottom margin
       int textDirection;                           // Sets text direction
       struct RTF_TABLEBORDER_FORMAT borderLeft;    // Cell RTF_TABLEBORDER_FORMAT structure
       struct RTF_TABLEBORDER_FORMAT borderRight;   // Cell RTF_TABLEBORDER_FORMAT structure
       struct RTF_TABLEBORDER_FORMAT borderTop;     // Cell RTF_TABLEBORDER_FORMAT structure
       struct RTF_TABLEBORDER_FORMAT borderBottom;  // Cell RTF_TABLEBORDER_FORMAT structure
       bool cellShading;                            // Cell has shading
       struct RTF_SHADING_FORMAT SHADING;           // Cell RTF_SHADING_FORMAT structure
};
```

Use *marginLeft*, *marginRight*, *marginTop* and *marginBottom* members to set margins of the table cell. If this is not done, table cell margins are the same for the whole table. Using *textVerticalAligment* member text in the table cell can be written in different directions. These are values this member can take:

- RTF_CELLTEXALIGN_TOP, defines top cell text vertical alignment
- RTF_CELLTEXTALIGN_CENTER, defines center cell text vertical alignment
- RTF_CELLTEXTALIGN_BOTTOM, defines bottom cell text vertical alignment

Member *textDirection* can have following values:

- RTF_CELLTEXDIRECTION_LRTB, defines left to right and top to bottom text flow
- RTF_CELLTEXTDIRECTION_RLTB, defines right to left and top to bottom text flow
- RTF_CELLTEXTDIRECTION_LRBT, defines left to right and bottom to top text flow
- RTF_CELLTEXTDIRECTION_LRTBV, defines left to right and top to bottom vertical text flow
- RTF_CELLTEXTDIRECTION_RLTBV, defines right to left and top to bottom vertical text flow

If table cell has borders then following structure is defined:

```
struct RTF_TABLEBORDER_FORMAT
{
       bool border;                                 // Border exists
       struct RTF_BORDERS_FORMAT BORDERS;           // RTF_BORDERS_FORMAT structure
};
```

Members *borderLeft*, *borderRight*, *borderTop* and *borderBottom* from table cell formatting structure are of this type. If specified cell border exists *border* member of this struct is set to **true**. *BORDERS* member of this struct is of type of border formatting structure explained above.

In order to define cell shading *cellShading* member can take following values:

- RTF_CELLSHADINGTYPE_FILL, defines fill cell shading type
- RTF_CELLSHADINGTYPE_HORIZ, defines horizontal cell shading type
- RTF_CELLSHADINGTYPE_VERT, defines vertical cell shading type
- RTF_CELLSHADINGTYPE_FDIAG, defines forward diagonal cell shading type

- RTF_CELLSHADINGTYPE_BDIAG, defines backward diagonal cell shading type
- RTF_CELLSHADINGTYPE_CROSS, defines cross cell shading type
- RTF_CELLSHADINGTYPE_CROSSD, defines cross diagonal cell shading type
- RTF_CELLSHADINGTYPE_DHORIZ, defines dark horizontal cell shading type
- RTF_CELLSHADINGTYPE_DVERT, defines dark vertical cell shading type
- RTF_CELLSHADINGTYPE_DFDIAG, defines dark forward diagonal cell shading type
- RTF_CELLSHADINGTYPE_DBDIAG, defines dark backward diagonal cell shading type
- RTF_CELLSHADINGTYPE_DCROSS, defines dark cross cell shading type
- RTF_CELLSHADINGTYPE_DCROSSD, defines dark cross diagonal cell shading type

After having table cell formatted call *rtf_start_tablecell()* function (the function argument is table cell right margin in twips from the page left margin) to write table cell formatting to the RTF document. Each table cell must be closed by calling *rtf_end_tablecell()* function before calling again this function. If it is not done in this manner the behavior of the RTF viewer can not be determined.

Now, paragraph text can be added to table cell, but with *tableText* member of the paragraph formatting structure set to **true** in every paragraph that is added to table cell. When finished working with table set again this member to **false**.

To get merged cells right cell margin should match right margin of some other cell in the table (above or below that one). In this release nested tables are not supported.

Formatting document

In this part a few document formatting options will be discussed. Document formatting structure is shown below:

```
struct RTF_DOCUMENT_FORMAT
{
       int viewKind;          // Sets document view mode
       int viewScale;         // Sets document zoom level
       int paperWidth;        // Sets document paper width (the default is 12240)
       int paperHeight;       // Sets document paper height (the default is 15840)
       int marginLeft;        // Left margin in twips (the default is 1800)
       int marginRight;       // Right margin in twips (the default is 1800)
       int marginTop;         // Top margin in twips (the default is 1440)
       int marginBottom;      // Bottom margin in twips (the default is 1440)
       bool facingPages;      // Facing pages (activates odd/even headers and gutters)
       int gutterWidth;       // Gutter width in twips (the default is 0)
       bool readOnly;         // User can't edit document
};
```

viewKind member can be one of the following:

- RTF_DOCUMENTVIEWKIND_NONE, defines no document view mode
- RTF_ DOCUMENTVIEWKIND_PAGE, defines page document view mode
- RTF_ DOCUMENTVIEWKIND_OUTLINE, defines outline document view mode
- RTF_ DOCUMENTVIEWKIND_MASTER, defines master document view mode
- RTF_ DOCUMENTVIEWKIND_NORMAL, defines normal document view mode

viewScale member determine document zoom level. Also, members *paperWidth* and *paperHeight* determine page size in twips. To set page margins use *marginLeft*, *marginRight*, *marginTop* and *marginBottom* members. Define gutter size using *gutterWidth* member. If pages should be facing then set *facingPages* member to **true**. Finally, if document should be protected against editing set *readOnly* member to **true**.

Conclusion

In this document a freeware C-library for generating RTF document files from source code is presented. In this release following is supported:

- Document creating and formatting
- Section creating and formatting
- Paragraph creating and formatting
- Character formatting
- Images loading
- Tables creating and formatting
- Tabs
- Columns
- Bullets and numbering

Using methods explained in here it is possible to export data from custom application to Rich Text Formatted document. Than, that document should be easy to transfer to different platforms which have RTF reader applications.

No errors or missing parts are guaranteed only for MSWord97 to MSWord2003, in this release. Other RTF viewers may show generated document different then it really is (WordPad as an example).

License details

Author information

Nikolić Darko, application developer
Serbia & Montenegro
18000 Niš
elmeh@bankerinter.net