Getting started with

ScroogeXHTML for Object Pascal

Version 9.3

LIMITED WARRANTY

No warranty of any sort, expressed or implied, is provided in connection with the library, including, but not limited to, implied warranties of merchantability or fitness for a particular purpose. Any cost, loss or damage of any sort incurred owing to the malfunction or misuse of the library or the inaccuracy of the documentation or connected with the library in any other way whatsoever is solely the responsibility of the person who incurred the cost, loss or damage. Furthermore, any illegal use of the library is solely the responsibility of the person committing the illegal act.

Contents

Introduction

About ScroogeXHTML

Features

ScroogeXHTML for Object Pascal converts text attributes including background and highlight colors, paragraph attributes including alignment (left, right, centered, justified) and paragraph indent (left, right, first line) and simple numbered or unnumbered lists. Unicode conversion allows international documents, including simplified and traditional Chinese, Korean and Japanese. CSS and default font settings allow to create optimized documents.

Supported output document types:

- XHTML 1.0 Strict
- HTML5

Limitations

The RTF specification contains very many elements and features, the library converts a *limited subset*.

API Documentation

The API documentation can be found in the installation folder, a link to the current on-line version can be found at https://www.scroogexhtml.com/object_pascal.html

ScroogeXHTML for the Java™ Platform

The library is also available for the Java platform: https://www.scroogexhtml.com.

Installation

Requirements

ScroogeXHTML supports the following development environments:

- Delphi 2009 or newer
- Free Pascal 3.2.2 or newer

Installation in Delphi

Time saving tip Installing into a package is not required. The component works fine if you simply add the source path to your project. This will save you the work of installing it for every new version

To install the component in the component palette, follow these steps:

1. **Create a new Delphi Package Project**
   - in the IDE menu, select "File | New | Package - Delphi"
2. **Add the file ScroogeXHTML_reg.pas**
   - in the IDE menu, select "Project | Add to Project ..."
   - choose the file <Scrooge Install Folder>\source\ScroogeXHTML_reg.pas
   - click on Open
3. **Install the package**
   - in the project manager view, right-click on the package node
   - choose "Install"

The component will appear in a new palette page with the title 'Habarisoft'.

Conversion Methods

Main conversion methods

One low-level function (declared in the TSxMain class) and several high-level conversion methods (declared in the TCustomScrooge class) are contained:

Convert	This function converts a string or a TStream containing RTF code and returns a HTML output string (UTF8String).
ConvertRTFFile	This procedure converts an existing RTF file and saves the result HTML code in an output file
ConvertRTF	This function converts an existing RTF file and returns a HTML string (UTF8String)

Conversion example

This example shows how to use the Convert method to write the HTML code to standard output.

Code example

```
program Example1;
{$APPTYPE CONSOLE}
uses
  ScroogeXHTML;
var
  SX: TBTScroogeXHTML;
begin
  SX := TBTScroogeXHTML.Create;
  try
    WriteLn(SX.Convert('{\rtf1 {\b bold \i Bold Italic \i0 Bold
again}}'));
    ReadLn;
  finally
    SX.Free;
  end;
end.
```

The generated output:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>
      Untitled document
    </title>
  </head>
  <body>
    <p>
      <span style="font-weight:bold;">bold </span><span style="font-
weight:bold;font-style:italic;">Bold Italic </span><span style="font-
weight:bold;">Bold again</span>
    </p>
  </body>
</html>
```

This example shows that the default component settings will generate a document with the HTML5 document type and a head section with default values for title and meta tags.

Let's take a closer look at the result code:

- The first line contains the document type declaration.
  - The document type can be set with the DocumentType property.
- The `<head>` section contains a document title element.
  - The DocumentTitle property can be used to set its content.
- The `<meta charset>` tag will be generated automatically and tells the browser that the document uses the UTF-8 encoding.
- The body contains a `<p>` element which uses elements to apply styles to the text parts with bold and bold/italic format.

Important notes:

- It is possible to use the component to generate only the body part.
  - See section AddOuterHTML for more details.

Configuration Options

Document Related Properties

Document Type

The DocumentType property selects the output document type. There are two groups of output documents: HTML based and XHTML based.

HTML based

- HTML5
- HTML 4.01 Strict

By default, HTML5 will be used.

Font Related Properties

FontConversionOptions

The conversion of font size, font name and other character properties can be controlled with the FontConversionOptions property. This property is a set of flags:

opFontSize	enables conversion of font sizes
opFontName	enables conversion of font names. See also TCustomScrooge.ReplaceFonts
opFontStyle	enables conversion of font styles (bold, italic, underlined, strikeout)
opFontColor	enables conversion of font colors
opFontBGColor	enables conversion of background font colors
opFontHLColor	enables conversion of highlight font colors

By default all options are enabled. If you want to activate only a subset of these options, your code may assign a list of these options to the FontConversionOptions property (e. g. `[opFontSize, opFontName]`).

Example code

In this example, the FontConversionOptions are set to an empty list

Code example

```
program Example2;
{$APPTYPE CONSOLE}
uses
  ScroogeXHTML, SxTypes;
const
  EXAMPLE_RTF = '{\rtf1 {\b bold \i Bold Italic \i0 Bold again}}';
var
  SX: TBTScroogeXHTML;
begin
  SX := TBTScroogeXHTML.Create;
  try
    SX.OptionsHead.AddOuterHTML := False;
    SX.FontConversionOptions := [];
    WriteLn(SX.Convert(EXAMPLE_RTF));
  finally
    SX.Free;
  end;
  ReadLn;
end.
```

The resulting output does not include the font style tag

```
    <p>
      bold Bold Italic Bold again
    </p>
```

Font Name Replacement

If RTF documents use fonts which are not available in the HTML browser, the look of the converted document is unpredictable. As a workaround, the component offers a font name replacement option which uses a list of replacement rules.

The **ReplaceFonts** property contains these default replacement rules:

Arial	all font names starting with “Arial” will be replaced by “Arial,Helvetica,sans-serif”
Courier	all font names starting with “Courier” will be replaced by “Courier,monospace”
Symbol	all font names starting with “Symbol” will be replaced by “Symbol”
Times	all font names starting with “Times” will be replaced by “Times,serif”

The replacement rules are key-value pairs. The following example shows how a new set of replacement rules can be defined.

Code example

```
...
  SX.ReplaceFonts.Clear;
  SX.ReplaceFonts.Add('Arial=sans-serif');
  SX.ReplaceFonts.Add('Courier=monospace');
```

```
  SX.ReplaceFonts.Add('Times=serif');
...
```

Font Size Scale

The FontSizeScale property sets the font size scale. The following units are supported:

point (pt)	Font sizes are expressed in point (pt). This is the default property value. Point is an absolute size scale, all others are relative scales.
em	Relative font sizes, expressed in em units.
ex	Relative font sizes, expressed in ex units.
percent	Relative font sizes, expressed in percent values.

Special Character related Options

Conversion of Space Characters

In HTML browsers, there is no visual difference between a single space character and a sequence of two or more space characters.

Some RTF documents however make use of sequences of space characters for special visual effects, for example in combination with a fixed-space font like Courier.

These documents would look corrupt when they are converted to HTML. As a workaround, a Boolean property, **ConvertSpaces**, can be used to replace sequences of space characters by "nonbreakable spaces" (` `).

The ConvertSpaces property is set to False by default.

Tab characters

HTML does not support the "Tab" character. However, this character may appear in RTF documents.

As a workaround, the component replaces tab characters by a sequence of non breakable spaces. The **TabString** property can be used to modify the replacement string.

Paragraph related options

Empty Paragraphs

Set the **ConvertEmptyParagraph** property to True to replace empty paragraphs, where the opening tag is followed by the closing tag by a line break tag (
).

Indentation

Set the **ConvertIndent** property to True if you want to activate support for left and right paragraph indents.

The ConvertIndent property is set to False by default.

Note the right indent in the output document is relative to the browser window - if you change the browser window size, the text area will adjust its size

Advanced Configuration

Embedding HTML output in existing documents

In many cases, the result of the RTF to HTML conversion shall not be a stand-alone document but should be included in existing HTML code.

For example, the component may be used to convert RTF code which is stored in a database table to build one HTML document with all the converted table records. This will however not be easy if the resulting HTML contains the `<head>` and `<body>` elements – a HTML document can only contain one `<head>` and one `<body>` section.

A HTML document where we want to include generated HTML code somewhere in the body section, could look like this:

```
<html>
  <head>
    <title>
      Business report for 2021
    </title>
  </head>
  <body>
    <h1>
      Summary
    </h1>

... HTML generated by ScroogeXHTML should go here ...

  </body>
</html>
```

AddOuterHTML Property

The OptionsHead property group controls the generation of HTML head and body elements, including the document title, META tags, and CSS stylesheet settings.

The AddOuterHTML property controls if the output will be a standalone HTML document or just a HTML fragment for embedding.

- Set this property to True (the default value) to create output documents with surrounding tags for the HTML header and body
- Set this property to False to create output documents without surrounding tags for the HTML header and body

Code example

```
program Example3;
{$APPTYPE CONSOLE}
uses
  ScroogeXHTML;
const
  EXAMPLE_RTF = '{\rtf1 {\b bold \i Bold Italic \i0 Bold again}}';
var
  SX: TBTScroogeXHTML;
begin
  SX := TBTScroogeXHTML.Create;
  try
    SX.OptionsHead.AddOuterHTML := False;
    WriteLn(SX.Convert('{\rtf1 {\b bold \i Bold Italic \i0 Bold
again}}'));
  finally
    SX.Free;
  end;
  ReadLn;
end.
```

The generated output:

```
    <p>
      <span style="font-weight:bold;">bold </span><span style="font-
weight:bold;font-style:italic;">Bold Italic </span><span style="font-
weight:bold;">Bold again</span>
    </p>
```

This code now could be used to embed it in an existing HTML document.

Hyper Links

How Hyper Links are detected

The component can detect hyper links in two ways:

- A hyper link which is embedded in the RTF as a HYPERLINK field expression.[1]
  - In this case, the RTF field result will be used as the link destination (URL).
  - Both ConvertFields and ConvertHyperlinks must be set to True.
- If a piece of text is formatted blue and underlined (example: hyperlink)
  - In this case, the formatted text will be used 'as-is' as the link destination.
  - The component does not check if the blue and underlined text is a valid link address.

1 {\field{*\fldinst{HYPERLINK example.com }}{\fldrslt{example.com\ul0\cf0}}}

HyperlinkURIBuilder property Event Handler

The developer can assign a custom link URI builder to have control over the hyper link processing.

Code example

```
SX.HyperlinkURIBuilder := MyCustomLinkURIBuilder;
```

Create smaller documents

Default Font Styles

OptionsOptimize Property Group

The properties

- IncludeDefaultFontStyle
- DefaultFontName
- DefaultFontColor
- DefaultFontSize

are useful to create smaller HTML documents. The optimization method uses a CSS definition in the HEAD section, which defines the default font settings in the document. The component will only create font properties if a text block has a different style.

Example

A simple RTF document (for example, created with WordPad) uses 'Times,serif' 10 pt as the main font. With the default settings, the component will create the full CSS style definition for every text block:

```
   <p>
     <span style="font-family:Times,serif;color:#000000;font-
size:10pt;">Hello World</span>
   </p>
```

To define and use the default CSS definition, use the following code:

Code example

```
...

SX.OptionsOptimize.DefaultFontName  := 'Times,serif';
SX.OptionsOptimize.DefaultFontColor := '#000000';
SX.OptionsOptimize.DefaultFontSize  := 10;
SX.OptionsOptimize.IncludeDefaultFontStyle := True;
SX.OptionsHead.AddOuterHTML := True;

...
```

The HEAD section of the generated document will now look like this:

```
        <head>
          <title>
            Untitled document
          </title>
          <meta http-equiv="content-type" content="text/html;
charset=iso-8859-1">
          <style>
          <!--
            body (font-family:Times,serif;font-size:10pt;color:#000000; )
              -->
          </style>
        </head>
```

And the HTML code for the paragraph will now be reduced to:

```
    <p>
       Hello World
    </p>
```

Post processing

Introduction

ScroogeXHTML for Object Pascal supports to

- post-process the intermediate document before the final conversion

Requirements

To add post-processing, a class needs to be created which implements the IPostProcessEventListener interface:

Code example

```
  IPostProcessEventListener = interface
  ['{0CD2CED9-E78D-4E8F-AE14-C8B34C2E771E}']
    procedure PostProcess(Sender: TObject; const EventObject:
TPostProcessEventObject);
  end;
```

Implementing class example:

Code example

```
type
  TMyPostProcessListener = class(TInterfacedObject,
IPostProcessEventListener)
  public
    procedure PostProcess(Sender: TObject; const EventObject:
TPostProcessEventObject);
  end;
```

In the method, the EventObject contains a Document property which represents the intermediate document tree. This property can be used to iterate over all paragraph nodes, and within every paragraph, over the formatted text nodes.

The example below is taken from the demo application, and adds a special character (¶) to the end of every empty line.

Code example

```
procedure TMyPostProcessListener.PostProcess(Sender: TObject; const
EventObject: TPostProcessEventObject);
var
  Doc: ISimpleDomDocument;
  I, J: Integer;
  Node: TSimpleDomNode;
begin
  Doc := EventObject.Document;
  for I := 0 to Doc.GetCount - 1 do
  begin
    for J := 0 to Doc.Item[I].GetCount - 1 do
    begin
      Node := Doc.Item[I].Item[J];
      if Node.TextContent = '' then Node.Add('¶');
    end
  end;
end;
```

Finally, add the listener:

Code example

```
PPL := TMyPostProcessListener.Create;
SX.PostProcessEventListeners.Add(PPL);
```

Picture Support

Introduction

ScroogeXHTML for Object Pascal supports to

- collect raw picture data for embedded pictures
- access raw picture data as a stream

Requirements

To extract data for embedded pictures, two properties of the component need to be set:

ConvertPictures This Boolean property is False by default. The component will only convert pictures if it is set to true.

PictureAdapter This property is not assigned by default. A picture adapter implementation must be assigned.

The IPictureAdapter interface is described in the next chapter.

An example implementation is included with source code (TMemoryPictureAdapterDataURI).

IPictureAdapter Interface

The `IPictureAdapter` interface defines methods for extraction of picture data. Most methods in this interface are only required for the internal data processing. Usually, custom implementations only need the `getPictures` function.

getPictures Returns a dictionary with all embedded pictures

The code example below is taken from the demo application. It uses the `PictureAdapter.getPictures` function to retrieve the picture data and save it to disk.

Code example

```
procedure TfrmMain.startConversion(Sender: TObject);
var
  Pics: TObjectDictionary<string, TEmbeddedPicture>;
  Filename: string;
  E: TEmbeddedPicture;
  Path: string;
```

```
begin
  SetOptions;

  SX.ConvertRTFFile(Filelistbox1.FileName, ediHTMLFile.Text);

  if SX.ConvertPictures then
  begin
    Pics := SX.PictureAdapter.getPictures;
    Path := ExtractFilePath(ediHTMLFile.Text);
    for Filename in Pics.Keys do
    begin
      E := Pics[Filename];
      if E.IsDataURI then begin
        Continue;
      end;
      with TMemorystream.Create do
        try
          LoadFromStream(E.Stream);
          SaveToFile(Path + '\' + Filename);
        finally
          Free;
        end;
    end;
  end;

  ShellExecute(0, 'open', PChar(ediHTMLFile.Text), '', '', SW_SHOWNORMAL)
end;
```

Example

Code example

```
uses
  ScroogeXHTML, SxInterfaces, SxMemoryPictureAdapter, (...)
var
  SX: TBTScroogeXHTML;
  PictureAdapter: IPictureAdapter;
begin
  SX := TBTScroogeXHTML.Create;
  try
    PictureAdapter := TMemoryPictureAdapterDataURI.Create;
    SX.PictureAdapter := PictureAdapter;
    SX.ConvertPictures := True;

    HTML := SX.ConvertRTF(RtfFile);

    // now use PictureAdapter methods to access the pictures
    ...

  finally
    SX.Free;
  end;
end;
```

Picture Data Conversion

For custom picture format conversion, the `IPictureAdapter` interface provides an optional method, `SetConverter`, which takes a parameter of type `IPictureConverter`.

Code example

```
if SX.ConvertPictures then
begin
  MyPictureConverter := TmyPictureConverter.Create;
  SX.PictureAdapter := TmemoryPictureAdapterDataURI.Create;
  // use my custom picture converter
  SX.PictureAdapter.SetConverter(MyPictureConverter);
end;
```

International Support

Unicode and Code Pages

Unicode conversion allows international documents, including simplified and traditional Chinese, Korean and Japanese.

ScroogeXHTML for Object Pascal uses Unicode to create the output HTML document. This allows to include (and mix) many different languages in one HTML page.

Requirements on the Client Side

On the client side, this solution works only if the HTML client (browser) supports Unicode and the necessary Unicode-enabled fonts are available on the system.

Requirements on the Producer Side

Many RTF documents use contain Unicode encoded characters, which need no further processing in the converter component. They use the numeric Unicode values which can be translated immediately to HTML.

Some RTF documents however use national 'code pages' to encode special characters.

If the component runs on a computer system which has no support for these code pages installed, the conversion of these RTF documents will not work.[2]

Left-to-right and right-to-left text direction

The component supports both LTR and RTL text directions.

Language Attributes

The component supports language attributes. The ConvertLanguage property activates support for language conversion. The DefaultLanguage property sets the default language of the document.

2 Disclaimer: The component uses mappings between code pages and character sets which might be incomplete or wrong. There is no guarantee that the code page conversion always works as expected

Logging

Configuration of Log Messages

Usage of the OnLog event handler

To add logging, assign a log event handler to the property OnLog

Code example

```
type
  TMyClass = class
  ...
  private

  // declare log event handler
  procedure MyLogHandler(Sender: TObject;
                         const logLevel: TlogLevel;
                         const logText: string);

...
// assign the log event handler
constructor TMyClass.Create;
begin
  inherited;

  // instantiate ScroogeXHTML component
  SX := ...
  ...
  // set the log handler
  SX.OnLog := MyLogHandler;

end;


// implement logging
TMyClass.MyLogHandler(procedure(Sender: TObject;
  const logLevel: TlogLevel; const logText: string);
begin
  // write string to file, console or outputdebugstring
  ...
end;
```

Setting the log level

The `LogLevel` property can be used to control the detail level of the logging procedure.

Logging over other logging frameworks

ScroogeXHTML for Object Pascal supports SLF4P (Simple Logging Facade for Pascal) which can be enabled by defining the conditional symbol

SCROOGE_USE_SLF4P

If this symbol is defined, the source folder of SLF4P must be added to the project search path. Also, the project must include one unit to bind the facade with the actual logging framework.

SLF4P is available as Open Source on its project home page:

https://github.com/michaelJustin/slf4p

Validation

Detect missing RTF header

To detect a missing RTF header, set the property NoRTFHeaderAction to acRaiseException.

The default value of this property is acReturnEmptyString.

In both cases, in case of a missing RTF header the converter will write an error to the log if logging is enabled.

A valid RTF document starts with

```
{\rtf
```

Frequently Asked Questions

Conversion

Why are empty paragraphs not shown in the result page?

HTML browsers do not show empty/white space only <p> elements. Example:

RTF view

```
Line 1
Line 2

Line 3
```

will look different in the HTML browser

Browser view

```
Line 1
Line 2
Line 3
```

Solution:

To remove empty space between lines, try to set the **ConvertEmptyParagraph** property to True. The result HTML then will contain
 or
 instead of empty <p> elements.

How can I reduce the space between paragraphs?

You may use CSS to remove space between paragraphs.

Code example

```
scrooge.setStyleSheetInclude("body, p {\n"
            + "  margin: 0px;\n"
            + "}\n");
```

Breaking changes in version 9.0

Removed Elements

Document Types

These output document types were deprecated and have been removed or commented out:

- XHTML Basic 1.0
- XHTML Mobile Profile 1.0
- XHTML 1.0 Transitional
- XHTML 1.1
- HTML 4.01 Strict and Transitional
- HTML 3 Flex

Changed Requirements

Compiler requirements

- Minimum supported Free Pascal version is now 3.2.2

Minor Changes

Moved units

All units which are experimental, complimentary or unsupported have been moved to the new folder `source/optional`.

MemoryPictureAdapterDataURI

The threshold for switching to data URI has been set to 1 kilobyte. Bigger pictures will not converted to data URIs.

Moved types and constants

Several types and constants related to embedded pictures moved from unit `SxTypes` to `SxEmbeddedPicture`.

Deprecation List

- Blue and underlined text will no longer be used for hyper link detection in a future release.

Breaking changes in version 8.0

Removed Elements

Removed properties

- AbortConversion
- ConvertUsingPrettyIndents
- ElementClasses
- ElementStyles
- HyperlinkOptions
- OnAfterConvert
  - superseded by PostProcessEventListeners property
- OnBeforeConvert
  - superseded by PostProcessEventListeners property
- OnProgress
- OnHyperlink
  - superseded by HyperlinkURIBuilder property
- OptionsHead.MetaOptions

Removed classes

- ScroogeXHTMLVCL
  - code for TRichEdit conversion can now be found in the Demo application source code
- TMemoryPictureAdapter
  - superseded by TMemoryPictureAdapterDataURI

Removed variables

- SxConst.SCROOGE_INDENT_CHAR
  - superseded by MarginBuilder property

Index

Reference