Rich Text to HTML Text Conversion

Twm Davies

Published by the Symbian Developer Network

Version: 1.0a – October 2007

1 INTRODUCTION **2**

2 BACKGROUND **3**

2.1 PRIMER ON TEXT FORMATTING AND STORAGE 3

2.1.1 CRichText *5*

2.1.2 CParaFormat *5*

2.1.3 TCharFormat *5*

3 RTTOHTML **5**

3.1 CONVERTING RICH TEXT TO HTML 6

3.1.1 Enumerating paragraphs *6*

3.1.2 Processing paragraphs *7*

3.1.3 Processing characters *7*

3.1.4 Adding char format processors *8*

3.2 CONVERTING HTML TO RICH TEXT 9

3.2.1 Test driven development *10*

3.2.2 Round tripping *11*

3.2.3 Using Picture embedding for HTML tags *11*

3.2.4 Processing the HTML *13*

3.3 OTHER STUFF 13

3.3.1 HTML Entities *13*

3.3.2 Packaging the converter as a DLL *14*

4 EXAMPLE APPLICATION **16**

5 SUMMARY **18**

1 Introduction

While writing MobileMind – an application which reads Freemind XML documents – I was faced with an engineering problem. The PC-based Freemind application has support for embedding formatted text by allowing the user to insert HTML markup into a text field, but the content is just stored as the user typed it. I also found that the use of embedded HTML is not an esoteric use case and that many Freemind users paste HTML from other applications.

In order to support as many real world FreeMind documents as possible in MobileMind, I had to come up with a scheme for supporting the display and editing of the HTML.

My requirements were:

- WYSIWYG editing of the most common formatting attributes
- Accurate importing and exporting of HTML
- Round-tripping of HTML so that pasted HTML from any source is preserved when loading and saving in MobileMind documents

Since there is no HTML editor control provided by Symbian UI platforms, I needed to come up with a scheme that would handle the requirements.

The end result was a generic DLL which offers services to convert HTML to Symbian's internal rich text format and back again. Figure 1 shows how MobileMind uses the DLL to take arbitrary HTML from a FreeMind Mindmap and present it to the user for editing. Note that the basic formatting (bold etc.) is present, but the more complex elements such as tables are left as tags, allowing the user to either edit around them or remove the tags completely.

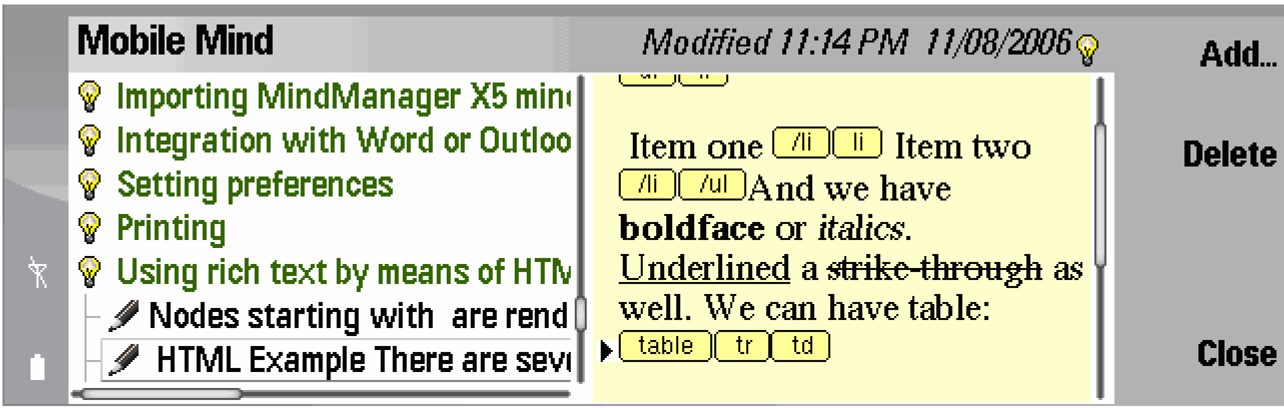


Figure 1: MobileMind demonstrating the HTML editor

This document describes in detail the creation of the DLL. It describes how I solved the problem of converting HTML to rich text in order to facilitate editing using the stock rich editor text control and then describes how the contents are externalised in HTML. Along the way, it also describes the Symbian OS 9.x text storage system and shows how to interrogate and manipulate the content.

The target audience of this paper is developers wishing to gain a deeper understanding of the Symbian text classes and those wishing to build upon this work. After reading the paper and studying the code, a developer should have a pretty good idea of how to write rich text in a XML format and read it back in again.

The code which accompanies this article is the full source for the HtmlRichTextConverter.dll and a simple UI driver application for S60 3.x and UIQ 3.x, to illustrate the DLL in use.

2 Background

Rich text is a generic term often used to describe text with extensive formatting and mark-up applied. In the context of this document, rich text refers to Symbian's native storage of formatted text, which has little in common with Microsoft RTF (rich text format) documents.

For a smartphone, Symbian OS offers a fairly comprehensive set of classes for storing, manipulating and displaying rich text. This generous support derives from the origins of Symbian OS as the software for the *Psion Series 5*. Not only did the *Series 5* have a capable built-in word processor but it was also possible to insert and edit rich text elsewhere, for example in calendar notes and jotter entries. With Symbian's emphasis on re-use, the effort to create a simple text editor on the Symbian platform boils down to just a few lines of code - allowing 3rd party application developers to stick a rich text editor control in their application with just a couple of resource file definitions.

Symbian OS stores rich text in a compact binary format which can be externalised and internalised from a store or a file. Unfortunately since the format is proprietary, it adds an obstacle to pre-generate content or to externalise the rich text into other formats.

The following section details the storage mechanism and introduces the key APIs and classes involved in text storage and manipulation. These APIS will be used in the next section by the conversion code.

2.1 Primer on text formatting and storage

The best way to visualise the storage of rich text on Symbian is that all the characters which make up the document (including line feeds and new paragraph markers) are stored in an dynamic array ranging from [0:`DocumentLength()`]. Any index into this array is called a document position.

Edward Bernays	Paragraph 0 Index[0:14]
Overview	Paragraph 1 Index[15:23]
Born in Vienna, Bernays was both a blood nephew and a nephew-in-law to Sigmund Freud, the father of psychoanalysis. Bernays's public relations efforts helped popularize Freud's theories in the United States. Bernays also pioneered the PR industry's use of psychology and other social sciences to design its public persuasion campaigns. "If we understand the mechanism and motives of the group mind, is it not possible to control and regiment the masses according to our will without their knowing about it? The recent practice of propaganda has proved that it is possible, at least up to a certain point and within certain limits." (*Propaganda*, 2005 ed., p. 71.) He called this scientific technique of opinion-molding the "engineering of consent."	Paragraph 2 Index[24:771]

Figure 2 Typical rich text document
(source: http://en.wikipedia.org/wiki/Edward_bernaise)

Figure 2 above shows how a typical rich text document is split into paragraphs. The fragment has three distinct paragraphs, the “Overview” heading and the body text with the occasional word highlighted in blue or italic.

The layout information for the text is stored in one or more text formatting layers. These are built on top of each other, adding further refinement to the text.

The top level formatting is achieved on a per paragraph level. “Overview” is a paragraph with a particular style, and the rest is a normal paragraph with certain ranges of characters containing additional styling information to add emphasis.

The kinds of attributes which are specified for the whole paragraph are things like text spacing, alignment and indentation and typeface appearance. A style is simply a pre-set collection of these attributes stored in a single named instance. For instance the overview section above may be based on the paragraph style “heading”.

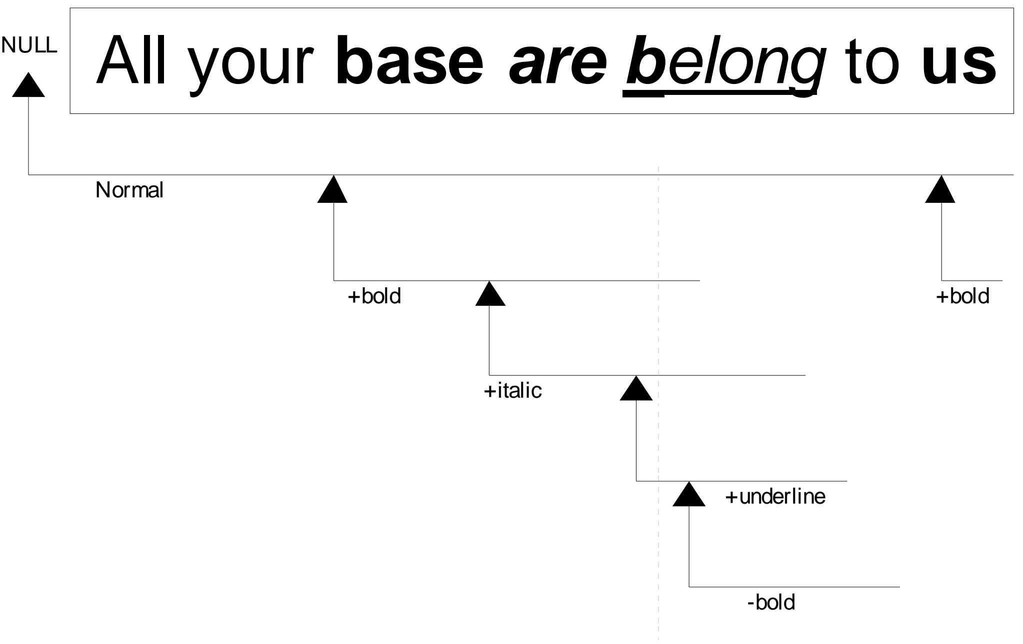


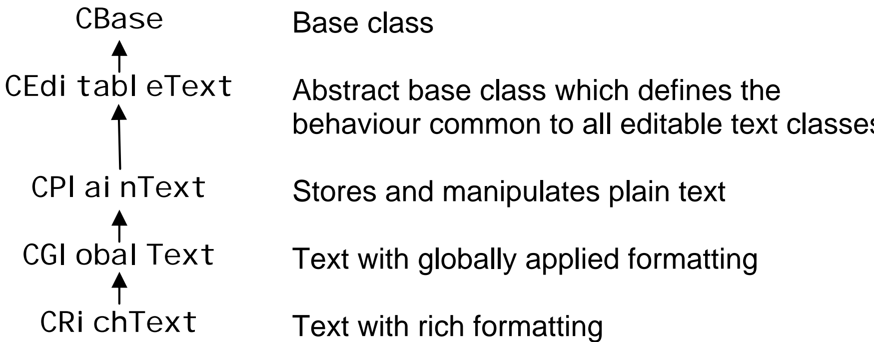
Figure 3: "Based on" layer chaining

Figure 3 shows how separate formatting objects are created using a based-on scheme to refine the formatting without duplicating information. Here the paragraph is based on the Normal style and each subsequent character formatting layer is based on that layer, adding one refinement in style.

In order to determine the exact formatting required for each character, each link in the formatting chain needs to be examined, and it’s formatting accumulated with its parent links until the parent is NULL.

2.1.1 CRichText

For the most part, most users of rich text APIs will be using a `CRichText` class which represents the key interface for manipulating rich text. `CRichText` is derived as follows:



`CRichText` provides many textual facilities, from adding new paragraphs to setting the style of a range of characters. To facilitate this, two formatting classes are derived from `CRichText`, `CParaFormat` and `TCharFormat`.

2.1.2 CParaFormat

This class represents the formatting applied to a current paragraph. `CParaFormat` objects are used when setting paragraph attributes for a range of text and are populated with attributes when retrieving the format of a particular paragraph. This class can be used to set or detect things like style, alignment or line spacing. `CParaFormat` applies to the whole paragraph.

2.1.3 TCharFormat

This class contains information about a section of text. When applying formatting to a particular range of characters, an instance of `TCharFormat` is used to set bold, italic, colour and so on. `TCharFormat` is also used when retrieving information about a text range. It can be used to detect if a given range has bold, italic etc. `TCharFormat` can apply to any range of text from a single character to a whole document.

`TCharFormat` comprises a `TFontSpec` for font dependent attributes and a `TFontPresentation` for font independent attributes.

3 RtToHTML

The main architecture of our example, RtToHtml, consists of a `CHtmlToRichTextConverter` class which holds the main body of both the rich text parser and the HTML parser. The parsing within the class is fairly light and most of the style processing is delegated to an array of `TCharFormatProcessors`. The processors are used for writing HTML when the formatting of a text fragment matches the style that the processor deals with, and used for reading with the HTML tag is matched. Figure 4 shows the class diagram of the component. Each class will be introduced during the rest of this section.

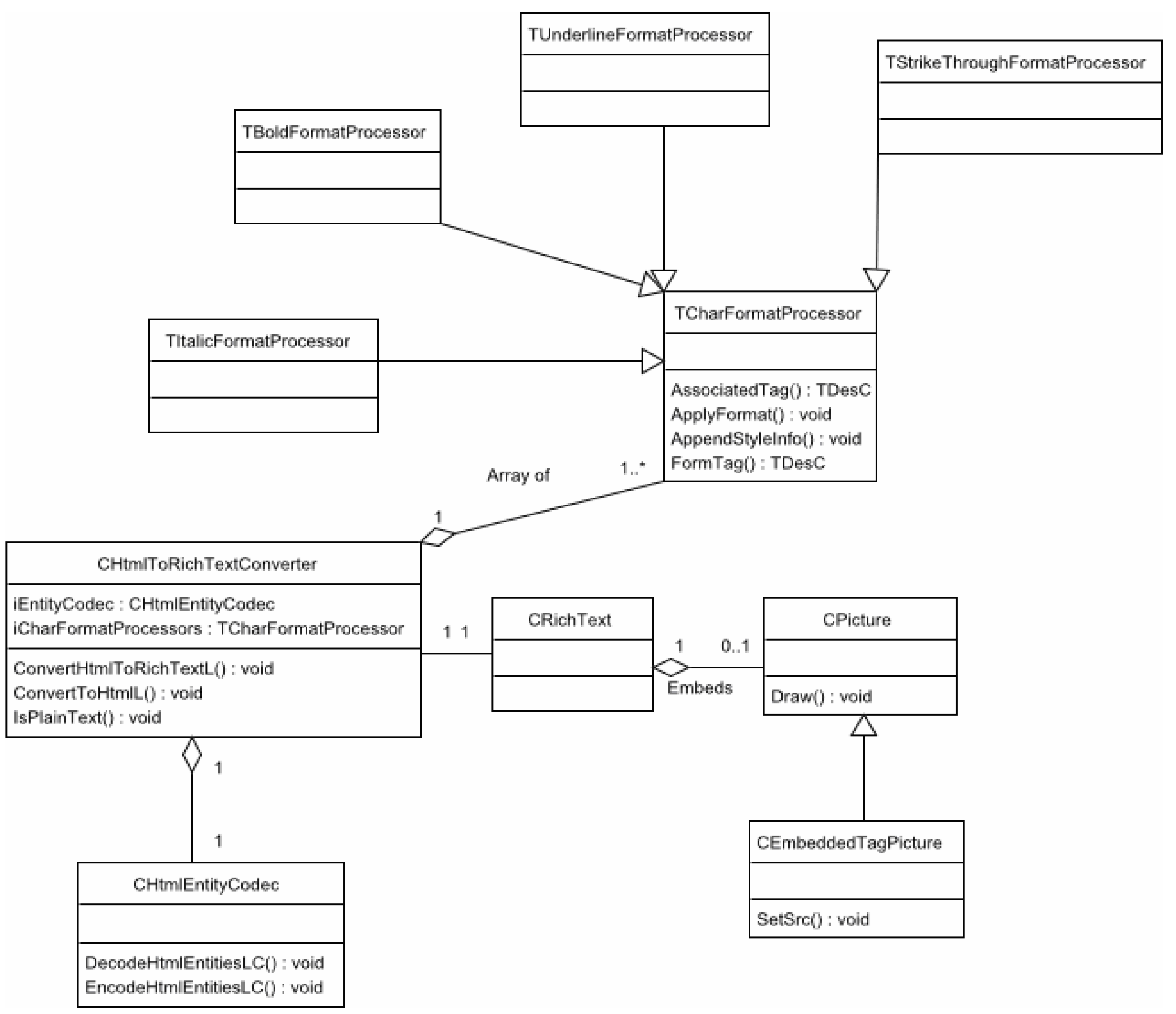


Figure 4: UML Class Diagram of the RtToHtml classes

3.1 Converting rich text to HTML

Since rich text is stored as a series of character and paragraph layers, the final formatting of each character has to be inspected in order to determine what HTML must be applied.

3.1.1 Enumerating paragraphs

The first task is to divide and conquer the task of converting to HTML by splitting the rich text into individual paragraphs.

`CRichText` can be used to enumerate paragraphs by returning the character and position of a numbered paragraph within the text as in the following code fragment:

```
for(TInt paraNum=0; paraNum<aRt.ParagraphCount(); paraNum++)
    {
    TInt paraLength=0, paraStart=0;
    paraStart = aRt.CharPosOfParagraph(paraLength, paraNum);
    Encoder_ProcessEachParagraphL(aRt, paraStart, paraLength);
    }
```

aRt is a reference to a CRichText object.

3.1.2 Processing paragraphs

In HTML, paragraphs are represented by a <p> tag to denote the start of the paragraph and a </p> to denote the end.

The parser loop starts each paragraph with a <p> and inspects the CParaFormat to decide if there are any attributes supported for the current paragraph. The only attribute currently considered is the alignment.

```
CParaFormat* paraFormat = CParaFormat::NewLC();
aRt.GetParagraphFormatL(paraFormat, aStart);
// aStart is the start of the paragraph identified by enumerating them

_LIT(KHtml_P_CENTER,"<p align=\"center\">");
_LIT(KHtml_P_RIGHT,"<p align=\"right\">");
_LIT(KHtml_P_DEFAULT,"<p>");

switch(paraFormat->iHorizontalAlignment)
    {
    case CParaFormat::ECenterAlign:
        aBuf.Append(KHtml_P_CENTER);
        break;
    case CParaFormat::ERightAlign:
        aBuf.Append(KHtml_P_RIGHT);
        break;
    default:
        aBuf.Append(KHtml_P_DEFAULT);
    }
```

Once the paragraph alignment is known, the text and character formatting that makes up the paragraph must be formed (see next section); finally the paragraph is closed with </p> and the parser moves on to the next paragraph.

3.1.3 Processing characters

Within a single paragraph, the processing of character formats (e.g. bold, italic etc) requires a little thought since each character has to be inspected to determine the final formatting for that character.

A brute force approach would be to iterate through each character, opening and closing a new HTML style for each character attribute. i.e. “hello” rendered in bold would be output as:

```
<b>h</b><b>e</b><b>l</b><b>l</b><b>o</b>
```

This would lead to bloated HTML and slow parsing and is not the prettiest HTML to have been generated. A better solution is to keep track of bands of common formatting.

One way of keeping track is to iteratively use the GetCharFormat() method of CRichText, taking note of the TCharFormatMask parameter, aVaries.

If GetCharFormat() is called on a range of text and the formatting does not change in the range (e.g. all characters in the range are bold and italic) then aVaries will be NULL, If one character in the range is bold, but the rest normal then aVaries will indicate that the bold character format varies over the range.

This method is incredibly useful for detecting formatting bands.

The following pseudo code shows roughly how it can be used:

```
start = 0
offset = 1

FOREVER
    richtext.GetCharFormat(charFormat, formatVaries, offset)
    IF NOT formatVaries then
        outputHtmlForRange(start, offset)
        start = start+offset // move to next formatting band
        offset = 1
    ELSE
        offset++ // grow the current range
END
```

Consider a fragment such as this:

Hello I'm a *very* ***bold*** piece of *text.*

Once the ranges are established, it's reasonably trivial to convert them to fairly efficient HTML, as demonstrated by the following table.

Formatting range	"Hello I'm a "	"*very* "	"***bold*** "	"piece of "	"*text*"
Char format styles	Normal	Normal + italic	Normal + bold + italic	Normal	Normal + italic
HTML to output	Hello I'm a	<i>very </i>	<i>bold </i>	piece of	<i>text</i>

The above ranges will be converted to the following

```
Hello I'm a <i>very </i> <i><b>bold <i></b> piece of <i>text</i>
```

It's not perfectly efficient since the <i>very </i> <i>bold<i> could be expressed as <i>very bold</i>. But taking Occam's razor to the problem – the benefits of ensuring absolute efficiency in this case are not worth the effort of complicating the algorithm.

3.1.4 Adding character format processors

RtToHtml was designed so that when a range of common formatting is identified, it passes the range to each of the registered character processors in order for them to add their own styling information. This makes it very easy to add processors to support additional attributes such a font colour.

The example below shows how the Italic format processor works. AppendStyleInfo() is given a TCharFormat object which represents a band of uniform formatting. The italic format processor can decide if it wants to contribute to the output HTML for this band of formatting and does so by checking to see if the font's posture is italic.

```
class TItalicFormatProcessor : public TCharFormatProcessor
    {
public:
    const TDesC& AssociatedTag()
        {
        _LIT(KTag, "I");
        return KTag;
        }

    void AppendStyleInfo(TDes& aHTMLStream, const TCharFormat& aFormat,
                                             THtmlTagType aTagType)
        {
        if(aFormat.iFontSpec.iFontStyle.Posture() == EPostureItalic)
            {
            FormTag(aHTMLStream, aTagType);
            }
        }

    void ApplyFormat(TCharFormat& aFormat, TBool aSet)
        {
        TFontPosture posture = aSet ? EPostureItalic : EPostureUpright;
        aFormat.iFontSpec.iFontStyle.SetPosture(posture);
        }
    };
```

AppendStyleInfo() is called twice. Once with aTagType == EHtmlTagTypeOpen and once for EHtmlTagTypeClose,

The call to FormTag() uses a base class helper method to create the string <i> or </i> using a virtual call to AssociatedTag().

3.2 Converting HTML to rich text

Unlike XHTML which is highly structured, general HTML – the kind humans write - is very hard to process reliably. It's always a challenge to consider all the unstructured cases which may be thrown at the parser and the result is often an un-maintainable, organically growing mess of a parser which developers are afraid to modify.

In some ways, writing a HTML parser is like writing a game with artificial intelligence. In game development, you create creatures which have some level of interaction with the player and during the development phase, whenever you are not happy with the response, you add a new bit of logic (a few more if-then-else statements) which gives a satisfactory response, and then you try again with a slight variation in behaviour to see what the creature does, and so on until all permutations of interaction have been exhausted and the creature seems to exhibit some form of intelligence. A HTML parser tends to evolve in a similar way, where the stimulus is a variety of real world HTML sources, collected for their beautiful malformed uniqueness.

How much effort you spend on a HTML parser really depends on your application's needs and the source data.

However the one thing that will keep you sane is to start with the test suite, collect all the representative HTML that you would like to support and then start writing the parser after you have considered all possibilities that you would like to support fully.

3.2.1 Test driven development

Test driven development is where the test and APIs are written first, then the implementation. When writing any sort of parser, the test suite is fundamental and should be written first.

The test framework for RtHtml is a simple console app using `RTest`, where the MMP file pulls in the .cpp files from the main DLL and uses the internal functions directly. The advantage of this approach is that you can unit test individual functions which are not intended to be exported from the final DLL and it allows you to write the test code before the DLL is frozen.

```
LOCAL_C void ValidateHTMLL(const TDesC& aHTMLFragment,
                           const TDesC& aExpectedResult = KNullDesC)
    {
    CParaFormatLayer* paraFormatLayer = CParaFormatLayer::NewL();
    CleanupStack::PushL(paraFormatLayer);
    CCharFormatLayer* charLayer = CCharFormatLayer::NewL();
    CleanupStack::PushL(charLayer);
    CRichText* richText =  CRichText::NewL(paraFormatLayer, charLayer);
    CleanupStack::PushL(richText);

    HtmlToRichTextConverter::ConvertHtmlToRichTextL(*richText, aHTMLFragment);

    const TInt KMaxHtmlBufferSizeInBytes = 1000;
    TBuf<KMaxHtmlBufferSizeInBytes> htmlOut;
    HtmlToRichTextConverter::ConvertToHtmlL(*richText, htmlOut);

    if(!aExpectedResult.Length())
        {
        test(htmlOut.CompareF(aHTMLFragment)==KErrNone);
        }
    else
        {
        test(htmlOut.CompareF(aExpectedResult)==KErrNone);
        }

    CleanupStack::PopAndDestroy(3, paraFormatLayer);
    }
```

Most of the test cases are input HTML which are converted to rich text and back again. The function `ValidateHTMLL(aHTMLFragment, aExpectedResult)` converts the input HTML into rich text and compares it with the expected output. This tests both conversion directions in one step. If no expected output is specified, then it compares the result with the input.

With `ValidateHTMLL()` in place it's then just a matter of collecting enough test cases. The test cases should be like a zoo. You should collect strange and curious perversions of HTML from the wild and throw them at your code.

HTML fragments can easily be added to the test code, though it became apparent as the project evolved that the test cases should be placed in a separate text file and the test updated to read the cases from the file.

```
LOCAL_C void ImportSomeHtmlL()
    {
```

```
ValidateHTMLL(_L("<html><p><i><b>Hello</i></b><b>,
                                                      i&apos;m bold.</b></p></html>"));
ValidateHTMLL(_L("<html><p><i><b>Hello</i></b><b>, i&apos;m
      bold.</b></p><p><i><b>Hello</i></b><b>, i&apos;m bold.</b></p></html>"));
ValidateHTMLL(_L("<html><h1>A Heading</h1><p>A <b>normal</b>
                                                               paragraph</p></html>"));
ValidateHTMLL(_L("<html><h1>Some  double  spaced  text</h1>
                                            <p>A <b>normal</b> paragraph</p> </html>"),
                  _L("<html><h1>Some double spaced text</h1>
                                            <p>A <b>normal</b> paragraph</p></html>"));
ValidateHTMLL(_L("<html><p><UNKNOWNTAG>Hello, this tests unknown
                                                       tags</UNKNOWNTAG></p></html>"));
ValidateHTMLL(_L("h"),_L("<html><p>h</p></html>"));
ValidateHTMLL(_L("<html><p><b>tag</b> <i>after</i> tag leaves space in between?
                                                                       </p></html>"));
ValidateHTMLL(_L("<html>Hello <b>this</b> <i>is</i> some <i>rendered</i>
                                                <strike>XMTL</strike>HTML<br></html>"),
              _L("<html><p>Hello <b>this</b> <i>is</i> some <i>rendered</i>
                                           <strike>XMTL</strike>HTML<br></p></html>"));
}
```

The chief reason for collecting test cases is that when you change the parser to cope with certain HTML fragment, the tests will highlight regressions in handling HTML. If you fix the problems in all test cases before moving on to the new support, your parser will continue to get better. The growing test suite allows you to experiment with re-factoring as the algorithm gets cumbersome.

It's also very valuable to capture test cases for features which you don't support in order to verify that the cases are handled gracefully without leaking memory.

3.2.2 Round tripping

As discussed in the introduction, a key use case for RtToHtml is to import arbitrary HTML, allow the user to edit some parts of the formatting (bold, italic, underline), and to preserve the HTML so that when it is exported it matches the original HTML as closely as possible.

The general idea is to try to support anything that can be represented by the CRichText format class, but anything else must be preserved.

Trying to store the unsupported attributes in sync with the main text would be incredibly difficult and fragile, so I needed a better solution.

3.2.3 Using Picture embedding for HTML tags

Symbian's CRichText supports object embedding. It has long been possible to insert a Sketch drawing into a Word object. A picture inserted into rich text can be any class derived from CPicture. The fact that a client of CRichText can insert a derived class means that there is no limit on the amount of data which can be embedded. In the HTML case, if a tag which cannot be represented by rich text is detected during the parsing of the HTML, then a CEmbeddedTagPicture is inserted into the text. This object stores the tag name and associated attributes (i.e. for an image tag, the whole string of '<img src="moo.jpg">' is stored in the CEmbeddedTagPicture object).

```
void CHtmlToRichTextConverter::Decoder_InsertTagRepresentationImageL(
                 CRichText& aRichText, const TDesC& aImage, THtmlTagType aTagType)
    {
```

```
if(aImage.Length()==0)
    return;

CEmbeddedTagPicture* picture;
picture = new(ELeave) CEmbeddedTagPicture(TSize(0,0));
CleanupStack::PushL(picture);

HBufC* fullSrcBuf = HBufC::NewLC(aImage.Length()+1);
fullSrcBuf->Des()=KNullDesC;
if(aTagType==EHtmlTagClosing)
    fullSrcBuf->Des().Append(KHtmlCloseChar);

fullSrcBuf->Des().Append(aImage);
picture->SetSrc(*fullSrcBuf);
picture->SetCloseTag( (aTagType==EHtmlTagClosing) );

CleanupStack::PopAndDestroy(fullSrcBuf);

TPictureHeader header;
header.iPicture = TSwizzle<CPicture>(picture);

TInt pos = aRichText.DocumentLength();
aRichText.InsertL(pos, header);
CleanupStack::Pop(picture);
}
```

The `CEmbeddedTagPicture` object stores a complete tag or closing tag. So for instance an image would be stored in one tag as `<IMG src=”source.jpg”>`.

`CEmbeddedTagPicture` is also responsible for drawing the tag when the rich text is displayed or printed.

The tag is drawn as a rounded yellow rectangle with the title of the tag.

```
void CEmbeddedTagPicture::Draw( CGraphicsContext& aGc,
                                const TPoint& aTopLeft,
                                const TRect& /*aRect*/,
                                MGraphicsDeviceMap* aMap) const
    {
    TRect bitmapRect=aMap->TwipsToPixels(TRect(TPoint(),iSize-TPoint(5,0)));
    bitmapRect.Move(aTopLeft);

    aGc.UseFont(iFont);
    aGc.SetPenColor(KRgbBlack);
    aGc.SetBrushStyle(CGraphicsContext::ESolidBrush);
    aGc.SetBrushColor(TRgb(255,255,128));
    aGc.DrawRoundRect(bitmapRect, TSize(4,4));

    aGc.SetBrushStyle(CGraphicsContext::ENullBrush);

    const TPtrC tagName = TagName(Src());

    TInt baseline = bitmapRect.Height() /2 + iFont->AscentInPixels()/2;
    aGc.DrawText(tagName,bitmapRect,baseline,CGraphicsContext::ECenter,1);
```

```
        aGc.DiscardFont();
        }
```

The round tripping works surprisingly well considering the simplicity of the solution e.g. the source HTML may have tables in it and even though the rich text will not draw the table, the text in each cell can be viewed and edited by the user. The user can also remove tags simply by deleting the character.

3.2.4 Processing the HTML

The basic job of processing HTML is to split the content into tags and text payload. The main lexer loop searches for text between '<' and '>' and processes these tags appropriately.

The following pseudo code tries to capture the basic flow:

```
index=0
index2=0
WHILE !eos
    char c = inputStream[index];
    IF c == '<' THEN
        Tag = StringBetween(c and '>')
        // keep building up charformat when encountering bold, Italic etc tags
        ProcessTag(tag, charFormat, paraformat)
    ELSE
        text = StringBetween(c and '<');
        insertTextWithFormat(text,charFormat)
        IF endOfParagraph THEN
            // finish off the text with alignment etc
            ApplyParagraphFormat(paraFormat)
END
```

If the tag is a “p”, then a new paragraph is inserted into the rich text, and the paragraph format object records the alignment (centre, right etc.). The remaining tags are passed to the appropriate character format processor to parse for building up the `charFormat` object.

As an example, the Italic Character processor will be called when an “I ” tag is matched and will be asked to set or remove it's style from the `aFormat` object.

```
void ApplyFormat(TCharFormat& aFormat, TBool aSet)
    {
    TFontPosture posture = aSet ? EPostureItalic : EPostureUpright;
    aFormat.iFontSpec.iFontStyle.SetPosture(posture);
    }
```

3.3 Other considerations

3.3.1 HTML Entities

Any text content within the HTML may contain characters which are otherwise used for mark-up, e.g. '**<**' and '**>**' characters. HTML has made provision for this by using escape codes called entities for representing ambiguous text.

Character	*Entity*
>	>
<	<
“	"
'	'

When converting from HTML to rich text, the entities must be replaced by the correct character at the point the text is being inserted. When writing HTML, the characters must be replaced by the entity code.

This is a simple substitution provided by the class `CHtmlEntityCodec`.

```
class CHtmlEntityCodec : public CBase
    {
public:
    CHtmlEntityCodec();

    HBufC* DecodeHtmlEntitiesLC(const TDesC& aTextSection);
    HBufC* EncodeHtmlEntitiesLC(const TDesC& aTextSection);
    const TDesC& EntityForChar(TChar aChar);

private:
    enum {KNumberOfEntities=4};
    const TDesC* iEntityTokens[KNumberOfEntities];
    };
```

The only real consideration is that the escaped text may be larger than the unescaped text, due to single characters being replaced by a 4-6 character entity. The worse (but unlikely case) is that the output buffer needs to be 6 times the size of the input to avoid buffer overflow.

3.3.2 Packaging the converter as a DLL

The RtToHtml library can be used by several applications and components and so is packaged as a DLL.

3.3.2.1 Headerfile export

Firstly any public header files should be exported to the `\epoc32\include` tree by adding a PRJ_EXPORTS section to the bld.inf

```
PRJ_EXPORTS
..\inc\HtmlRichTextConverter.h
\epoc32\include\TwmDesign\RtHtml\HtmlRichTextConverter.h

PRJ_MMPFILES
HtmlRichTextConverter.mmp

PRJ_TESTMMPFILES
TestSuite.mmp
```

When packaging a DLL, It is important to choose a unique location for your exported headers. It's best not to export to \epoc32\include\, as an existing header file of the same name may be overwritten, causing clashes with other libraries on the SDK.

The header file defines the API, including relevant data structures and which functions can be called from outside the DLL.

It is always best to export only a minimal, complete interface which represents the API but does not expose the implementation, or functions used by the implementation. Only export to \epoc32\include\ what you would be happy to support in the future..

In this case I decided to expose three static functions which do three distinct tasks:

- convert to HTML
- convert from HTML and
- a function to determine if a rich text object needs to be exported as HTML or represented as plain Unicode text

```
#include <TXTRICH.H>

#ifndef __HTMLTORICHTEXTCONVERTER_H
#define __HTMLTORICHTEXTCONVERTER_H

namespace HtmlToRichTextConverter
{
    IMPORT_C void ConvertHtmlToRichTextL(CRichText& aRt, const TDesC& aHtmlStream);
    IMPORT_C void ConvertToHtmlL(CRichText& aRt, TDes& aHtmlStream);
    IMPORT_C TBool IsPlainText(CRichText& aRt);
}

#endif // __HTMLTORICHTEXTCONVERTER_H
```

The choice of namespace is important to avoid clashes with the global name pool and is the correct way of packaging a bunch of related static functions. It's fairly common on Symbian OS to use a class with only static functions to wrap functions as a pseudo name space. This works ok but you don't get any of the benefits of C++ name space management and importing.

3.3.2.2 Avoid binary name clash

On EKA1, private DLLs which were only used by a single application could be stored in "\system\apps\yourappname". When loading the main application binary, the DLL loader would check that path first before looking in "\system\libs".

Since Symbian OS v9 and the introduction of platform security, all binaries must reside in "\sys\bin" even if they are private to the application. This leads to a situation where it's possible for an application developer to choose a DLL name which clashes with an existing Symbian/S60/UIQ binary, or one which clashes with another 3rd party.

To avoid this, Symbian suggests placing the UID of the DLL in the DLL's name, for instance HtmlRichTextConverter(A1234567).dll. This is pretty ugly and it's probably fine just to put a vendor name in all your binaries. It's certainly better to be safe than sorry since at any time in the future,

Symbian or one of the UI vendors may put a DLL or .EXE which clashes with yours, meaning that your application will no longer install on the latest devices.

3.3.2.3 Freezing

During the development of a DLL, the DEF file must be left unfrozen by placing a `exportsunfrozen` directive in the MMP file. This serves to indicate that the API definition is unstable and stops anyone linking against the DLL. Once the DEF file is complete the API is frozen by removing `exportsunfrozen` from the MMP file and calling "`abld freeze`" from the command line or "Freeze def files" from the Carbide c++ IDE.

The result of this is an "`HtmlRichTextConverter.def`" file for the target in the project tree and an import LIB ready for clients of the DLL to link against. Once the LIB file has been published, care must be taken to preserve binary compatibility – once another application or DLL relies on the exported APIs the DLL maintainer must only add new functions to the end of the DEF file.

4 Example application

The test app is very simple: it presents a rich text editor which is initially populated by converting a HTML string literal stored in the code into rich text by using the DLL. It calls `HtmlToRichTextConverter::ConvertHtmlToRichTextL()`.

From a UI point of view, the important class is the control which provides the editing and display of rich text. `CEikRichTextEditor` (a `CCoeControl`) uses `CRichText` for storage and allows display and editing and supports cut and paste if required.

Figure 5 shows the screen when launching the test application on both a S60 and a UIQ-based device.

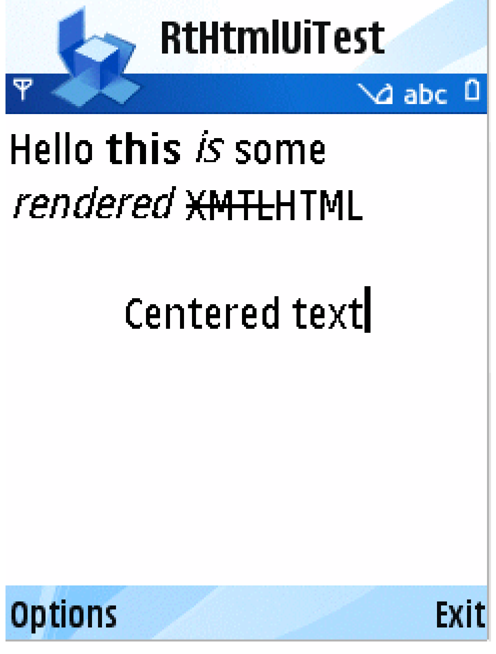


Figure 5: The Test App on launch on both UIQ and S60

The user can then enter text and use the menu, select a range of text using the shift and cursor keys and set the formatting by choosing bold/italic/underline etc from the menu, as shown in Figure 6.

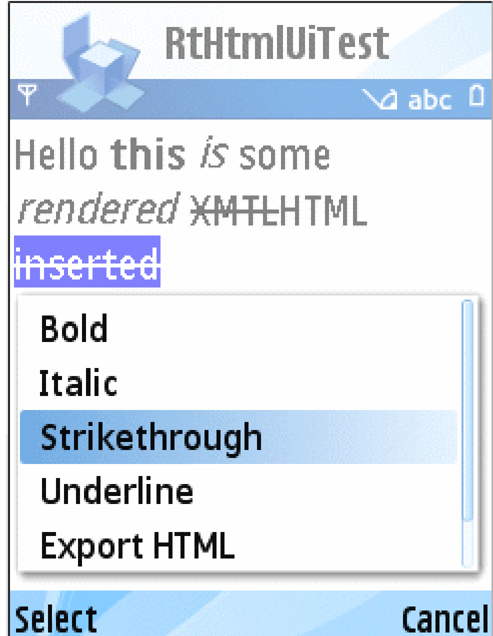
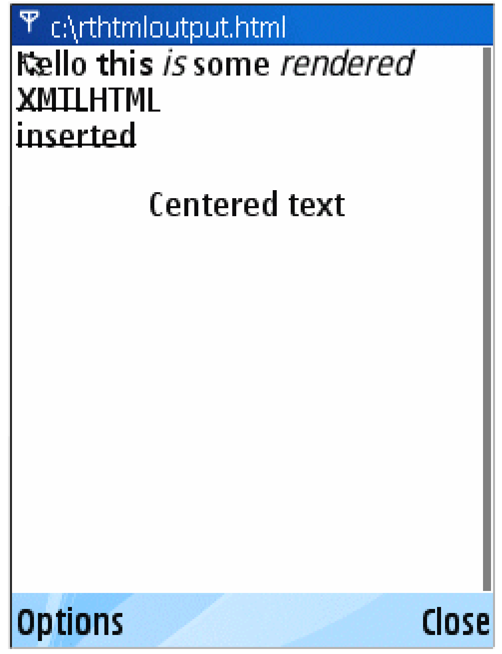


Figure 6: Menu options in example application on S60 device

The final option is to view the document in the web browser, providing a quick way of verifying the conversion to HTML by visual inspection. Selecting "Export HTML" writes the rich text to a HTML file. On S60, the HTML can be viewed by using the File Manager to navigate to "`C:\data\others\rthtml output.html`" and opening the file.

On UIQ, the browser is automatically launched when "export HTML" is selected.



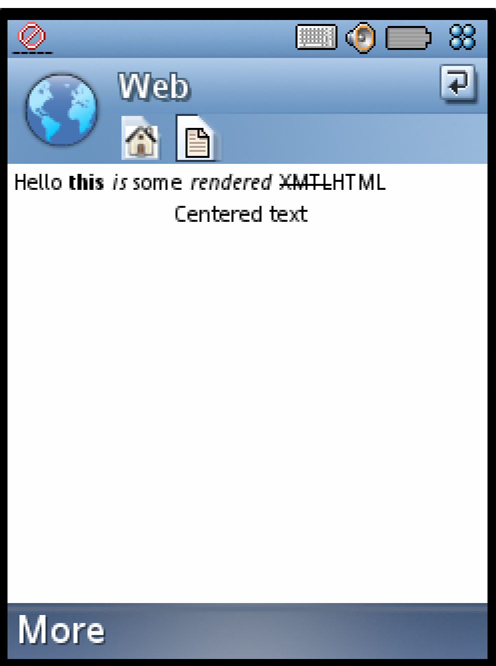


Figure 7: The HTML viewed in the default browser on S60 3.0 and UIQ 3.0

5 Summary

I have presented a basic method for round tripping HTML and Symbian rich text. The implementation as-is provides fairly minimal support for HTML, but represents most of the important mark-up available to a user entering rich text into an application. I've stressed that the test suite is important in the development and using the test suite as a guard for refactoring the algorithm is the fundamental principle in evolving the parser.

About the Author

Twm Davies works as an independent consultant in the Symbian sphere. He graduated from Cardiff University with a first in computer science, joining Symbian in 1999 where he worked for almost eight years. He initially worked as a developer of the 'crystal' messaging application which provided the UI to the Nokia communicator range and later specialised in consulting on handset stability and performance, assisting in taking many familiar Symbian devices to market.

Twm is a contributing author to the recently published "Symbian OS Communications Programming" book from Symbian Press and is the author of MobileMind - mind mapping software for S80-based devices (http://www.twmdesign.co.uk/mobilemind/).

Want to be kept informed of new articles being made available on the Symbian Developer Network?

Subscribe to the Symbian Community Newsletter.

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.