```
//ÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐ
//  FILENAME:   eText.TaggedText.m
//  SUMMARY:    Implementation of Tagged markup formats of eText (HTML, LaTeX)
//  CATEGORY:   TaggedText
//  PROTOCOLS:  NXRegisterPrintfProc()
//  INTERFACE:  See ChooseEncoding.Tool
//  AUTHOR:     Rohit Khare and Tom Zavisca
//  COPYRIGHT:  ©1993,94 California Institute of Technology, eText Project
//ÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐ
//  Implementation Comments
//      There's a lot of malarkey involved with the API for the encoders and
//  the printfProc registrations. All the "user" needs is + flushHTMLEncoding.
//
//      currentHTMLEncoding is a file-global char** of ENTITIES entries
//      currentHTMLEncodingLength is a file-global unsigned char[ENTITIES];
//      defaultHTMLEncoding is a char*[ENTITIES] C array.
//      An attempt is made to read encodings from the file specified in
//  a user dwrite.
//
//      Something that bothers me about encoders: how can we properly use
//  Symbol font?  Σ ≠ \sigma ?
//
//      After investigating the interactions of Annotations and Tagging,
//  I have decided that it is not neccessary to return to ground state
//  before writing HTML for an annotation. The physical and logical tags
//  apply to the run containing the annotation as well.
//      Reason: link button on same line as an H3 font descrip.
//      Output: close, linkbutton, open, desc, close splits into two lines.
//ÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐ
//  History
//  11/04/94:  DESIGN REV: HTML annotations don't start from ground state
//  10/17/94:  Cleaned up for eText5.
//  08/05/94:  Completely Rearchitected for 5.0. RK
//ÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐ
//  Imported Interfaces
//
    #import "eText.TaggedText.h"
    #import <ctype.h>

@implementation eText(TaggedText)
```

```
//ÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐ
//  Stream Operators
//
- writeHTML:(NXStream *) s withTags:(taggingInfo *) tags {
    int                     k,N;
    NXRun                   *curr;
    NXTextBlock             *currBlock;
    int                     currentOffset,targetOffset;
    NXAtom                  closer;
    taggingInfo             *aTag,*found;
    taggingInfo             fakeTag;
    id                      fm;          //FontManager

    N = theRuns->chunk.used/sizeof(NXRun);
    curr = theRuns->runs;
    currBlock = [self firstTextBlock];
    currentOffset = 0;
    closer = NULL;
    fm = [FontManager new];
```

```
for (k=0; k < N; k++) {
    if (curr->info == NULL) {
        // Encode the state for this run.
        // First, is this a tagged run? If so, do we need to close the
        // previous state?

        aTag = tags; found=NULL;
        while (!found && aTag && aTag->font) {
            if (curr->font == aTag->font) found = aTag;
            aTag++;
        }

        if (!found) {             // search for physical tags
            char tmp[32],*family;
            NXFontTraitMask traits;
            int weight;
            float size;
            BOOL isFixedPitch;

            *tmp = 0;
            isFixedPitch =          // primitive monospacing test
                ([curr->font metrics])->isFixedPitch;
            [fm getFamily:&family traits:&traits weight:&weight
                size:&size ofFont:curr->font];
            if (isFixedPitch) strcat(tmp, "<TT>");
            if (curr->rFlags.underline) strcat(tmp, "<U>");
            if (traits & NX_BOLD) strcat(tmp, "<B>");
            if (traits & NX_ITALIC) strcat(tmp, "<I>");
            if (*tmp) {             // don't bother unless we got styles
                fakeTag.start = NXUniqueString(tmp);
                *tmp = 0;
                if (traits & NX_ITALIC) strcat(tmp, "</I>");
                if (traits & NX_BOLD) strcat(tmp, "</B>");
                if (curr->rFlags.underline) strcat(tmp, "</U>");
                if (isFixedPitch) strcat(tmp, "</TT>");
                fakeTag.end = NXUniqueString(tmp);
                found = &fakeTag;
            }
        }
```

```
    if (found) {                // stop previous tag, if differs
        if (found->end != closer) {
            if (closer) {       // transition
                NXWrite(s, closer, strlen(closer));
            }
            NXWrite(s, found->start, strlen(found->start));
            closer = found->end;
        }
    } else if (closer) {        // return to ground state
        NXWrite(s, closer, strlen(closer));
        closer = NULL;
    }
} else {
    // clear tagging state, write out the annotation
    // MAJOR DESIGN CHANGE: SEE HISTORY & NOTES!!!! RK, 11/4
    // if (closer) {
    //  NXWrite(s, closer, strlen(closer));
    //  closer = NULL;
    //}
    if ([curr->info respondsTo:@selector(writeHTML:forView:)]) {
        [curr->info writeHTML:s forView:self];
    }
}

// encode the text corresponding to the run
// misson is to write (cumulative) curr->chars chars beginning
// at currentCount. boundaries may map onto > 1 block
targetOffset = currentOffset + curr->chars;
// consume full blocks
while ((currBlock) && (targetOffset >= (currBlock->chars))) {
    if (!(curr->info))          // throw annotated bits in bucket
        if(targetOffset > currentOffset) // don't pass len=0 to encoder
            HTMLEncoder(s, currBlock->text+currentOffset,
                        currBlock->chars - currentOffset);
    targetOffset-=currBlock->chars;
    currBlock=currBlock->next;
    currentOffset=0;
}
// consume partial block
if (currBlock && (! curr->info))     // throw annotated bits in bucket
```

```
          if(targetOffset > currentOffset) // don't pass len=0 to encoder
              HTMLEncoder(s, currBlock->text + currentOffset,
                          targetOffset-currentOffset);
      currentOffset=targetOffset;
      curr++;
  }
  if (closer) {
      NXWrite(s, closer, strlen(closer));
      closer = NULL;
  }
  return self;
}

- writeLaTeX:(NXStream *) s withTags:(taggingInfo *) tags {
  int                     k,N;
  NXRun                   *curr;
  NXTextBlock             *currBlock;
  int                     currentOffset,targetOffset;
  NXAtom                  closer, oldStart=NULL;
  taggingInfo             *aTag,*found;
  taggingInfo             fakeTag;
  id                      fm;        //FontManager

  N = theRuns->chunk.used/sizeof(NXRun);
  curr = theRuns->runs;
  currBlock = [self firstTextBlock];
  currentOffset = 0;
  closer = NULL;
  fm = [FontManager new];

  for (k=0; k < N; k++) {
      if (curr->info == NULL) {
          // Encode the state for this run.
          // First, is this a tagged run? If so, do we need to close the
          // previous state?

          aTag = tags; found=NULL;
          while (!found && aTag && aTag->font) {
             if (curr->font == aTag->font) found = aTag;
             aTag++;
```

```
}

if (!found) {              // search for physical tags
    char tmp[32],*family;
    NXFontTraitMask traits;
    int weight;
    float size;
    BOOL isFixedPitch;

    *tmp = 0;
    isFixedPitch =         // primitive monospacing test
        ([curr->font metrics])->isFixedPitch;
    [fm getFamily:&family traits:&traits weight:&weight
        size:&size ofFont:curr->font];
    if (isFixedPitch) strcat(tmp, "{\\tt ");
    if (curr->rFlags.underline) strcat(tmp, "\\underline{");
    if (traits & NX_BOLD) strcat(tmp, "{\\bf ");
    if (traits & NX_ITALIC) strcat(tmp, "{\\it ");
    if (*tmp) {            // don't bother unless we got styles
        fakeTag.start = NXUniqueString(tmp);
```

```
            *tmp = 0;
            if (traits & NX_ITALIC) strcat(tmp, "}");
            if (traits & NX_BOLD) strcat(tmp, "}");
            if (curr->rFlags.underline) strcat(tmp, "}");
            if (isFixedPitch) strcat(tmp, "}");
            fakeTag.end = NXUniqueString(tmp);
            found = &fakeTag;
        }
    }

    if (found) {              // stop previous tag, if differs
        // how can we tell if the state has changed?
        // the assumption is that a run necessarily corresponds
        // to a change of style -- but a colorchange wouldn't.
        // with HTML, the exact closer string would be unique
        // for LaTeX the heuristic is that every _opener_ is unique
        // thus, if the opener is unchanged, we short-circuit the
        // close-reopen pair.
        if (found->start != oldStart) {
            if (closer) {     // transition
        NXWrite(s, closer, strlen(closer));
      }
      NXWrite(s, found->start, strlen(found->start));
      closer = found->end;
      oldStart = found->start;
    }
  } else if (closer) {   // return to ground state
    NXWrite(s, closer, strlen(closer));
    closer = NULL;
  }
} else {
  // clear tagging state, write out the annotation
  if (closer) {
    NXWrite(s, closer, strlen(closer));
    closer = NULL;
  }
  if ([curr->info respondsTo:@selector(writeLaTeX:forView:)]) {
    [curr->info writeLaTeX:s forView:self];
  }
}
```

```
// encode the text corresponding to the run
// misson is to write (cumulative) curr->chars chars beginning
// at currentCount. boundaries may map onto > 1 block
targetOffset = currentOffset + curr->chars;
// consume full blocks
while ((currBlock) && (targetOffset >= (currBlock->chars))) {
    if (!(curr->info))        // throw annotated bits in bucket
        if(targetOffset > currentOffset) // don't pass len=0 to encoder
            LaTeXEncoder(s, currBlock->text+currentOffset,
                        currBlock->chars - currentOffset);
    targetOffset-=currBlock->chars;
    currBlock=currBlock->next;
    currentOffset=0;
}
// consume partial block
if (currBlock && (! curr->info))      // throw annotated bits in bucket
    if(targetOffset > currentOffset) // don't pass len=0 to encoder
        LaTeXEncoder(s, currBlock->text + currentOffset,
                    targetOffset-currentOffset);
```

```
          currentOffset=targetOffset;
          curr++;
     }
     if (closer) {
          NXWrite(s, closer, strlen(closer));
          closer = NULL;
     }
     return self;
}

//ÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐ
//  String Operators
//
+ encodeHTML:(NXStream *) s from: (unsigned char *) theChars length: (int)len {
     HTMLEncoder(s,theChars,len); return self;}
+ encodeLaTeX:(NXStream *) s from: (unsigned char *) theChars length:(int)len {
     LaTeXEncoder(s,theChars,len); return self;}
+ encodeURI:(NXStream *) s from: (unsigned char *) theChars length:(int)len {
     URIEncoder(s,theChars,len); return self;}
```

```
//ÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐ
//  HTMLEncoder API
//
+ flushHTMLEncoding {
    // Next access will force reloading according to UserModel
    currentHTMLEncoding=NULL; return self;
}

@end

//ÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐÐ
//  Encoder API
//  note that these are file-globals, and thus apply to the
//  entire eText process; encodings are not chosen on a per-document basis.
//

const char     *defaultHTMLEncoding[ENTITIES];
char           **currentHTMLEncoding=NULL;
char           *currentBuffer;
unsigned char  *currentHTMLEncodingLength;
```

```
void HTMLEncoder(NXStream *stream, unsigned char *item, int len) {
    int i;

    if(!currentHTMLEncoding) {        // Hence the +flushHTMLEncoding
        char HTMLResourceFilePath[MAXPATHLEN];

        [[NXBundle mainBundle] getPath:HTMLResourceFilePath
                forResource:[userModel stringQuery:"HTMLEncoding"]
                ofType:ENCD_EXT];
        if(*HTMLResourceFilePath) {   // try to load from this path
            unsigned char  *tempBuffer;
            unsigned char  *tempLengths;
            char           **tempEncoding;

            tempBuffer = malloc(4*MAXPATHLEN*sizeof(unsigned char));
                          // The above is C Programmer's Disease
            tempLengths = malloc(ENTITIES * sizeof(unsigned char));
            tempEncoding = malloc(ENTITIES * sizeof(char *));
            if (readEncodingTableFromFile(
                HTMLResourceFilePath, tempEncoding, tempLengths, tempBuffer)){
                // no freeing if not defaultEncoding
                // is a memory leak noone cares about
                currentHTMLEncodingLength = tempLengths;
                currentHTMLEncoding = tempEncoding;
                currentBuffer = tempBuffer;
            } else {
                free(tempBuffer); free(tempLengths); free(tempEncoding);
                tempBuffer = tempLengths = tempEncoding = NULL;
                NXLogError("Could not read encoding data from %s",
                            HTMLResourceFilePath);
            }
        }
    }

    if (!currentHTMLEncoding) {       // Error fall-through
        // "use" the defaultEncoding.
        currentHTMLEncoding = defaultHTMLEncoding;
        currentHTMLEncodingLength=malloc(ENTITIES * sizeof(unsigned char));
        for(i=0;i<ENTITIES;i++)
```

```
         currentHTMLEncodingLength[i] =
             (currentHTMLEncoding[i] ? strlen(currentHTMLEncoding[i]) : 0);
    }

    // The two "modes" of the Encoder, using the userData parameter
    if(!len) len=strlen((unsigned char*)item);

    for (i=0; i<len; i++) {
        NXWrite(stream, currentHTMLEncoding[(unsigned char)item[i]],
                currentHTMLEncodingLength[(unsigned char)item[i]]);
    }
}

void URIEncoder(NXStream *s, unsigned char *item, int len) {
    int i;

    // The two "modes" of the Encoder, using the userData parameter
    if(!len) len=strlen((unsigned char*)item);

    for (i=0; i<len; i++) {
```

```
unsigned ch = item[i];
if (isalnum(ch) || ((ch=='%')&&isdigit(item[i+1])&&isdigit(item[i+2])))
    NXPutc(s,ch);
else switch (ch) {
    case ':':
    case '/':
    case '\\':
// "safe" in RFC1630 BNF
    case '$':
    case '-':
    case '_':
    case '@':
    case '.':
    case '&':
    case '+':
    // "extra" in RFC1630 BNF
    case '!':
    case '*':
    case '\"':
    case '\'':
          case '|':
          case ',':
                      NXPutc(s,ch); break;
          default:    // encode as %hex
                      NXPrintf(s,"%%%x", ch); break;
      }
    }
}

void LaTeXEncoder(NXStream *s, unsigned char *item, int len) {
    int i;

    // The two "modes" of the Encoder, using the userData parameter
    if(!len) len=strlen((unsigned char*)item);

    for (i=0; i<len; i++) {
        unsigned ch = item[i];
        switch (ch) {
            case '<':  NXWrite(s, "$<$",3); break;
            case '>':  NXWrite(s, "$>$",3); break;
            case '\\': NXWrite(s, "$\\backslash$",12); break;
            case '~':  NXWrite(s, "\\~",2); break;
            case '^':  NXWrite(s, "\\^",2); break;
            case '{':  NXWrite(s, "\\{",2); break;
            case '}':  NXWrite(s, "\\}",2); break;
            case '%':  NXWrite(s, "\\%",2); break;
            case '#':  NXWrite(s, "\\#",2); break;
            case '_':  NXWrite(s, "\\_",2); break;
            case '&':  NXWrite(s, "\\&",2); break;
            case '$':  NXWrite(s, "\\$",2); break;
            case '\n': NXWrite(s, "\\par\n",1); break;
            case '\t': // we should do something here for tabs
            default:   // we should do something here for extended symbols
                       NXPutc(s, ch); break;
        }
    }
}

BOOL readEncodingTableFromFile(const char *path, char **targetEncoding,
                               unsigned char *targetLengths,char*targetBuffer) {
```

```
NXStream    *s;
int         i,j,len,maxlen;
char        *theChars,*current;

s = NXMapFile(path, NX_READONLY);
if(s) {
    NXGetMemoryBuffer(s,&theChars,&len, &maxlen);
    i=j=0;
    while (i<ENTITIES && (j < len)) {
        while (theChars[j] == '#') { // consume comment lines
            while ((j<len) && (theChars[j++] != '\n'));
        }
        if (theChars[j++] == '\"'){  // we have a winner!
            current = targetBuffer;
            while (theChars[j] != '\"'){
                // heuristics identical to NXStringTable
                switch (theChars[j]) {
                    case '\\' :
                        switch (theChars[++j]) {
                        case 'n'  : *(targetBuffer++)= '\n'; break;
                        case 't'  : *(targetBuffer++)= '\t'; break;
                        case '\\' : *(targetBuffer++)= '\\'; break;
                        case '\"' : *(targetBuffer++)= '\"'; break;
                        case 'a'  : *(targetBuffer++)= '\a'; break;
                        case 'b'  : *(targetBuffer++)= '\b'; break;
                        case 'f'  : *(targetBuffer++)= '\f'; break;
                        case 'r'  : *(targetBuffer++)= '\r'; break;
                        case 'v'  : *(targetBuffer++)= '\v'; break;
                        default   : *(targetBuffer++)= theChars[j]; break;
                        } break;
                    default :   *(targetBuffer++) = theChars[j]; break;
                }
                j++;
            }
            *(targetBuffer++)=0;
            targetEncoding[i]=current;
            targetLengths[i]=strlen(targetEncoding[i]);
            i++;
        }
        while ((j<len) && (theChars[j++] != '\n')); // consume until EOL
```

```
        }
        NXCloseMemory(s, NX_FREEBUFFER);
        return YES;
    }
    return NO;
}


const char * defaultHTMLEncoding[ENTITIES] = {
    "",            /* NUL */
    "",            /* SOH */
    "",            /* STX */
    "",            /* ETX */
    "",            /* EOT */
    "",            /* ENQ */
    "",            /* ACK */
    "",            /* BEL */
    "",            /* BS */
    "\t",          /* TAB */
    "<BR>\n",      /* NEWLINE */
```

```
"",             /* VT */
"",             /* FF */
"\r",           /* CR */
"",             /* SO */
"",             /* SI */
"",             /* DLE */
"",             /* DC1 (XON)*/
"",             /* DC2 */
"",             /* DC3 (XOFF)*/
"",             /* DC4 */
"",             /* NAK */
"",             /* SYN */
"",             /* ETB */
"",             /* CAN */
"",             /* EM */
"",             /* SUB */
"",             /* ESC */
"",             /* FS */
"",             /* GS */
"",             /* RS */
"",               /* US */
" ",              /* SPACE */
"!",
"&quot;",
"#",
"$",
"%",
"&amp;",
"'",
"(",
")",
"*",
"+",
",",
"-",
".",
"/",
"0",
"1",
"2",
"3",
"4",
"5",
"6",
"7",
"8",
"9",
":",
";",
"&lt;",
"=",
"&gt;",
"?",
"@",
"A",
"B",
"C",
"D",
"E",
"F",
"G",
"H",
"I",
"J",
"K",
"L",
"M",
"N",
"O",
"P",
"Q",
"R",
"S",
"T",
"U",
"V",
"W",
"X",
"Y",
"Z",
"[",
"\\",
"]",
"^",
"_",
"`",
"a",
"b",
"c",
"d",
"e",
"f",
"g",
"h",
"i",
"j",
"k",
"l",
"m",
"n",
"o",
"p",
"q",
"r",
"s",
"t",
"u",
"v",
"w",
"x",
"y",
"z",
"{",
"|",
"}",
"~",
"",               /* DEL */
"&nbsp",
"&Agrave",
"&Aacute",
"&Acirc",
"&Atilde",
"&Auml",
"&Aring",
"&Ccedil",
"&Egrave",
"&Eacute",
"&Ecirc",
"&Euml",
"&Igrave",
"&Iacute",
"&Icirc",
"&Iuml",
"&ETH",        // 0x90
"&Ntilde",
"&Ograve",
"&Oacute",
"&Ocirc",
"&Otilde",
"&Ouml",
"&Ugrave",
"&Uacute",
"&Ucirc",
"&Uuml",
"&Yacute",
"&THORN",
"mu",
" x ",
" / ",
"(c)",          // 0xA0
" ! ",
" cents ",
" Pound ",
"/",
" Yen ",
" Florin ",
" Section ",
" Currency ",
"'",
"``",
"&lt;&lt;",
"&lt;",
"&gt;",
"fi",
"fl",
"(R)",         // 0xB0
"-",
" * ",         // dagger
" ** ",        // dubdagger
" . ",         // centered period
" | ",         // broken pipe
" P ",         // Paragraph
"*",           // bullet
",",           // low-quote
",,",          // low dubquote
"''",          // up dub
"&gt;&gt;",
"...",
"%%",          // per thousand
"!",           // not!
```

```
"?",           // upside down ?
"1",           // 0xC0
"`",
"'",
"^",
"~",
"-",           // macron
"\\/",         // breve
".",
"..",          // uml
"2",
"o",
",",           // cedilla
"3",
"''",
",",           // backward cedilla
"\\/",         // caron
"--",          // 0xD0
"+/-",
"(1/4)",
"(1/2)",
"(3/4)",
"&agrave;",
"&aacute;",
"&acirc;",
"&atilde;",
"&auml;",
"&aring;",
"&ccedil;",
"&egrave;",
"&eacute;",
"&ecirc;",
"&euml;",
"&igrave;",    //0xE0
"&AElig;",
"&iacute;",
"a",
"&icirc;",
"&iuml;",
"&eth;",
"&ntilde;",
"L",
"&Oslash;",
"OE",
"o",
"&ograve;",
"&oacute;",
"&ocirc;",
"&otilde;",
"&ouml;",
"&aelig;",
"&ugrave;",
"&uacute;",
"&ucirc;",
"i",
"&uuml;",
"&yacute;",
"l",
"&oslash;",
"oe",
    "B",
    "&thorn;",
    "&yuml;",
    "",
    ""
};
```