

```
//
// FILENAME: eTDocInfo.m
// SUMMARY: Required persistent interface for ALL eText documents
// SUPERCLASS: Object:HashTable:NXStringTable:eTDocInfo
// PROTOCOLS: <ComponentWriting,ComponentReading,InspectableTarget>
// INTERFACE: None
// AUTHOR: Rohit Khare
// COPYRIGHT: ©1993,94 California Institute of Technology, eText Project
//
// Implementation Comments
// eTDocInfos are descendants of StringTables,
// so they can be extended indefinitely to support dictionary
// attributes. Future applications may replace the string table
// with a bTree, allowing for more powerful indexing and the
// ability to browse eText document collections without loading
// all the .etDocInfos.
//
// HISTORY
// 12/23/94: Changed readComponent to cache the "latest" docPath
// 12/23/94: Overhauled taggingFont system to handle "missing" fonts
// 12/08/94: Patched Kelsey's HTML generation bugs.
// 10/30/94: Fixed tag font default registration, added +aliases for tags.
// 10/30/94: Modified to support <InspectableTarget>
// 10/29/94: search/regex package rewritten
// 10/27/94: -isVirgin added to public API.
// 10/04/94: -free bug; NXStrTab was free()ing NXAtoms. (was in init/rcfpth)
// 10/04/94: Added eTDoc ivar by analogy to path[] and eTComponents.
// 10/03/94: Added <ComponentWriting>, docInfoTable.
// 10/01/94: Reorganized into categories: eText and eDraw.
// 09/30/94: Revamped for eText5; renamed eTDocInfo
// 07/20/94: Added LaTeX support.
// 07/10/94: Added support for HTML header/trailer writing
// 01/11/94: Added Agent field.
// 01/11/94: Added flexible "bindery" methods for extension of navinfo data
// 01/10/94: Created. See Actors/eTNavinfo
//
// Imported Interfaces
//
// #import "eTDocInfo.h"
// #import "eTDocInfoUI.h"
// #import "Kludges.subproj/regexpr.h"
//
// "Class" Variable
//
// HashTable *docInfoTable;
//
// @implementation eTDocInfo
```

```
//
// Class Management
//
+ initialize {
    char key[MAXPATHLEN];

    docInfoTable = [[HashTable alloc] initWithKeyDesc:"i"];

    sprintf(key, "%s%s", H1, TAGNAME);

    [userModel defaultValue:"Times-Bold" for:key];
    sprintf(key, "%s%s", H1, TAGSIZE);
    [userModel defaultValue:"30.0" for:key];

    sprintf(key, "%s%s", H2, TAGNAME);

    [userModel defaultValue:"Times-Roman" for:key];
    sprintf(key, "%s%s", H2, TAGSIZE);
    [userModel defaultValue:"24.0" for:key];

    sprintf(key, "%s%s", H3, TAGNAME);
    [userModel defaultValue:"Times-Bold" for:key];
    sprintf(key, "%s%s", H3, TAGSIZE);
    [userModel defaultValue:"18.0" for:key];

    sprintf(key, "%s%s", H4, TAGNAME);
    [userModel defaultValue:"Times-Roman" for:key];
    sprintf(key, "%s%s", H4, TAGSIZE);
    [userModel defaultValue:"18.0" for:key];

    sprintf(key, "%s%s", H5, TAGNAME);
    [userModel defaultValue:"Times-Bold" for:key];
    sprintf(key, "%s%s", H5, TAGSIZE);
    [userModel defaultValue:"10.0" for:key];

    sprintf(key, "%s%s", H6, TAGNAME);
    [userModel defaultValue:"Times-Roman" for:key];
    sprintf(key, "%s%s", H6, TAGSIZE);
    [userModel defaultValue:"10.0" for:key];

    sprintf(key, "%s%s", QT, TAGNAME);
    [userModel defaultValue:"Helvetica" for:key];
    sprintf(key, "%s%s", QT, TAGSIZE);
    [userModel defaultValue:"14.0" for:key];
}
```

```

sprintf(key, "%s%s", BODY, TAGNAME);
[userModel defaultValue:"Times-Roman" for:key];
sprintf(key, "%s%s", BODY, TAGSIZE);
[userModel defaultValue:"14.0" for:key];

return self;
}
+ (HashTable *) docInfoTable {
    return docInfoTable;
}
+ findDocInfo:(long) aDocID {
    return [docInfoTable valueForKey:aDocID];
}

// Later, twiddling bits inside NXStrTab, I see that it is
// initialized to %-*, in contravention of the docs.
//
// For debugging purposes only: I suspect StrTab's munching me.
// Well, NXStringTable IS munching me!!!!
// If the current value is == NXUniqueString(aValue), it tries
// to free the old NXAtom!!!! So that triggers random lossage
// everywhere... the fix here is evil and expensive, so be it.
// NeXT engineering will burn for this (unless they have some
// extra-special magic on tap in Mecca...)
- (void *)insertKey:(const void *)aKey value:(void *)aValue
{
    NXAtom key,value;

    key = NXUniqueString(aKey);
    value = NXUniqueString(aValue);
    [self removeKey:key];
    return [super insertKey:key value:value];
}

- init {
    char    idStr[9], buf[MAXPATHLEN];
    struct passwd *p;

    [super initWithKeyDesc:"%" valueDesc:"%"];
    readOnly = NO;
    virgin = YES;
    docPath[0] = '\0';
    etDoc = nil;

    docID = 0;
    while ((docID == 0) || ([docInfoTable isKey:docID]))

```

[illegible]

```

    return self;
}
- readComponentFromPath:(NXAtom)path {
    char    docInfoPath[MAXPATHLEN]; id aDI;

    if ([docInfoTable valueForKey:docID] == self)
        [docInfoTable removeKey:docID]; // init: registered the untitled ID
    strncpy(docPath, path, MAXPATHLEN);

    // OK, we have an etfd path, try concatenating DOCINFOFILE.
    sprintf(docInfoPath, "%s/%s", docPath, DOCINFOFILE);
    if (access(docInfoPath, R_OK | F_OK)) {
        // backward-compatibility with PR1 documents.
        sprintf(docInfoPath, "%s/%s", docPath, NAVINFOFILE1);
        if (access(docInfoPath, R_OK | F_OK)) {
            // backward-compatibility with PR2 documents.
            sprintf(docInfoPath, "%s/%s", docPath, NAVINFOFILE2);
            if (access(docInfoPath, R_OK | F_OK)) {
                // Last-Ditch check: is path itself the file? [etApp openFile]
                strncpy(docInfoPath, path, MAXPATHLEN);
                strncpy(docPath, docInfoPath, MAXPATHLEN);
                *rindex(docPath, '/') = '\\0';
            }
        }
    }
    if (access(docInfoPath, R_OK | F_OK)) {
        [NXApp delayedFree:self];
        return nil;
    }
    if (![self readFromFile:docInfoPath]) {
        [NXApp delayedFree:self];
        return nil;
    }
    // Quick Patch for pre-5.0 documents
    if ([self isKey:@"title"] &&
        ([self docTitle] == NXUniqueString("Untitled Document")))
        [self setDocTitle:[self valueForKey:@"title"]];
    sscanf([self valueForKey:DOCID], "%x", &docID);
    aDI = [docInfoTable valueForKey:docID];
    if (!aDI){
        [docInfoTable insertKey:docID value:self];
    } else if (aDI != self){
        id realInfo;

        [NXApp delayedFree:self];
        realInfo = [docInfoTable valueForKey:docID];
        [realInfo setDocPath:path];
    }
}

```

[illegible]

```

- setType:(const char *) newType {
    [self insertKey:DOCTYPE value:NXUniqueString(newType)];
    return self;
}

- (NXAtom) author {
    return [self valueForKey:AUTHOR];
}

- setAuthor:(const char *) newAuthor {
    [self insertKey:AUTHOR value:NXUniqueString(newAuthor)];
    return self;
}

- (NXAtom) agent {
    return [self valueForKey:AGENT];
}

- setAgent:(const char *) newAgent {
    [self insertKey:AGENT value:NXUniqueString(newAgent)];
    return self;
}

- (long) parentID {
    long pID; int success;

    success = sscanf([self parent], "%x", &pID);
    return (success ? pID : 0);
}

- (NXAtom) parent {
    return [self valueForKey:PARENT];
}

- setParent:(const char *) newParent {
    [self insertKey:PARENT value:NXUniqueString(newParent)];
    return self;
}

- setParentID:(long) newParentID {
    char newParent[64];

    sprintf(newParent, "%x", newParentID);
    [self insertKey:PARENT value:NXUniqueString(newParent)];
    return self;
}

- (NXAtom) peers {
    return [self valueForKey:PEERS];
}

- setPeers:(const char *) newPeers {
    [self insertKey:PEERS value:NXUniqueString(newPeers)];
}

```

```

    return self;
}
- addPeerID:(long) newPeerID {
    char newPeers[MAXPATHLEN];           // Long enough? C Programmer's Disease.

    strncpy(newPeers, [self peers], MAXPATHLEN);
    sprintf(newPeers, " %x", newPeerID);
    [self insertKey:PEERS value:NXUniqueString(newPeers)];
    return self;
}

- (NXAtom) keywords {
    return [self valueForKey:KEYWORDS];
}
- setKeywords:(const char *) newKeywords {
    [self insertKey:KEYWORDS value:NXUniqueString(newKeywords)];
    return self;
}

- (NXAtom) rtfComments {           // RTF source of comment field
    return [self valueForKey:COMMENTS];
}
- setRTFComments:(const char *) newComments {
    [self insertKey:COMMENTS value:NXUniqueString(newComments)];
    return self;
}

- (const char *) comments {       // ASCII contents of comment field
    static char      comments[MAXPATHLEN];
    NXStream         *stream;
    id                theText = [NXApp sharedText];
    const char        *rtfSrc;

    rtfSrc = [self rtfComments];
    stream = NXOpenMemory(rtfSrc,strlen(rtfSrc), NX_READONLY);
    [theText readRichText:stream];
    NXClose(stream);
    [theText getSubstring:comments start:0 length:MAXPATHLEN-1];
    comments[MAXPATHLEN-1]=0;
    return comments;
}

- (NXAtom) lastModifyDate {
    return [self valueForKey:MDFYDATE];
}

- (NXAtom) valueOf: (NXAtom) theQuery {
    NXAtom tmp = [self valueForKey:theQuery];

```


[illegible]


```
[self insertKey:TRAILER value:(newState ? "YES" : "NO")];  
return self;  
}  
  
- (BOOL)emitTrailer {  
    NXAtom defVal = [self valueForKey:TRAILER];  
    if (!defVal || !strcmp(defVal, "YES")) return YES;  
    return NO;}  
  
//DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  
// Tagging Fonts  
//  
- initWithTagFields {  
    int i;  
    NXAtom tags[8] = {H1, H2, H3, H4, H5, H6, QT, BODY};  
    for(i=0; i<8; i++)  
        [self setTag:tags[i] to:[self tagFont:tags[i]]];  
    [self setEmitTrailer:YES];  
    return self;  
}  
  
- setTag:(const char *)tagType to: (Font *)newFont {  
    char     tagName[MAXPATHLEN];  
    char     tagSize[MAXPATHLEN];  
    char     fontSize[MAXPATHLEN];  
  
    sprintf(tagName, "%s%s", tagType, TAGNAME);  
    sprintf(tagSize, "%s%s", tagType, TAGSIZE);  
    sprintf(fontSize, "%f", [newFont pointSize]);  
    [self bindKey:NXUniqueString(tagName) to:NXUniqueString([newFont name])];  
    [self bindKey:NXUniqueString(tagSize) to:NXUniqueString(fontSize)];  
    return self;  
}  
  
// The following table is used to cache errors in tagFont and defaultTagFont.  
static id errTab = nil;  
  
- tagFont:(const char *)tagType {  
    char      tagName[MAXPATHLEN];  
    char      tagSize[MAXPATHLEN];  
    const char *fontName;  
    const char *fontSizeStr;  
    float     fontSize;  
    id         theFont;  
  
    sprintf(tagName, "%s%s", tagType, TAGNAME);  
    sprintf(tagSize, "%s%s", tagType, TAGSIZE);  
    fontName = [self valueOf:tagName];  
    fontSizeStr = [self valueOf:tagSize];  
    if ((*fontName && *fontSizeStr)) {
```

```

        fontName = [userModel stringQuery:tagName];
        fontSizeStr = [userModel stringQuery:tagSize];
    }
    sscanf(fontSizeStr, "%f", &fontSize);
    theFont = [Font newFont:fontName size:fontSize matrix:NX_FLIPPEDMATRIX];
    if (!theFont) {
        if (!errTab) {errTab = [[HashTable alloc] initWithKeyDesc:@"%"];}
        if (![errTab isKey:fontName]){
            NXLogError("Can't find %s font for the %s doc tag!", fontName, tagType);
            [errTab insertKey:fontName value:nil];
        }
        // Can we figure out if the fonts were supposed to be italic, bold, etc?
        theFont = [Font newFont:"Courier" size:fontSize matrix:NX_FLIPPEDMATRIX];
    }
    return theFont;
}

- registerTag:(const char *)tagType to: (Font *)newFont {
    return [eTDocInfo registerTag:tagType to:newFont];}
+ registerTag:(const char *)tagType to: (Font *)newFont {
    char    tagName[MAXPATHLEN];
    char    tagSize[MAXPATHLEN];
    char    fontSize[MAXPATHLEN];

    sprintf(tagName, "%s%s", tagType, TAGNAME);
    sprintf(tagSize, "%s%s", tagType, TAGSIZE);
    sprintf(fontSize, "%f", [newFont pointSize]);
    [userModel bindKey:tagName to:[newFont name]];
    [userModel bindKey:tagSize to:fontSize];
    return self;
}

- defaultTagFont:(const char *)tagType {
    return [eTDocInfo defaultTagFont:tagType];}
+ defaultTagFont:(const char *)tagType {
    char    tagName[MAXPATHLEN];
    char    tagSize[MAXPATHLEN];
    const char *fontName;
    const char *fontSizeStr;
    float    fontSize;
    id        theFont;

    sprintf(tagName, "%s%s", tagType, TAGNAME);
    sprintf(tagSize, "%s%s", tagType, TAGSIZE);
    fontName = [userModel stringQuery:tagName];
    fontSizeStr = [userModel stringQuery:tagSize];
    sscanf(fontSizeStr, "%f", &fontSize);
    theFont = [Font newFont:fontName size:fontSize matrix:NX_FLIPPEDMATRIX];

```

```
if (!theFont) {  
    if (!errTab) {errTab = [[HashTable alloc] initWithKeyDesc:@"%"];}  
    if (![errTab isKindOfClass:[NSString class]]){  
        NXLogError("Can't find %s font for the %s default!",fontName,tagType);  
        [errTab insertObject:fontName forKey:nil];  
    }  
    // Can we figure out if the fonts were supposed to be italic, bold, etc?  
    theFont =[Font fontWithName:@"Courier" size:fontSize matrix:NX_FLIPPEDMATRIX];  
}  
return theFont;  
}
```

//XX

```
// Alternate Formats  
//  
- writeHTMLHeader:(NXStream *)s // required  
{  
  
    char tmp[MAXPATHLEN];  
    long aDocID;  
    id theDocInfo;  
    id docInfoList;  
    NXAtom peers;  
    int i;
```

NXPrintf(s,"<HEAD>\n<TITLE>%v</TITLE>\n",[self docTitle]);

```
strcpy(tmp,[self parent]);  
sscanf(tmp,"%x",&aDocID);  
theDocInfo=[docInfoTable valueForKey:aDocID];  
if(theDocInfo){  
    strcpy(tmp,[theDocInfo docPath]);  
    *rindex(tmp,'.')=0;  
    NXPrintf(s,<"LINK HREF=\"%V.\"HTMD_EXT\"/"HTML_INDEX\"" REL=\"Parent\""  
TITLE=\"%v\">\n", rindex(tmp,'/')+1,[theDocInfo docTitle]);  
}
```

docInfoList = [[NXApp navigator] query:[self docIDStr field:PARENT];
i = [docInfoList count];
for (;i;i--) {
 strcpy(tmp,[docInfoList objectAtIndex:i-1] docPath));
 *rindex(tmp,'.')=0;
 NXPrintf(s,<"LINK HREF=\"%V.\"HTMD_EXT\"/"HTML_INDEX\"" REL=\"Child\""
TITLE=\"%v\">\n", rindex(tmp,'/')+1,[[docInfoList objectAtIndex:i-1] docTitle]);
}

```
// if ([self parent] != NXUniqueString("")) {  
    docInfoList = [[NXApp navigator] query:[self parent field:PARENT];
```

```

        i = [docInfoList count];
        for (;i;i--) {
            if ([docInfoList objectAtIndex:(i-1)] != self) {
                strcpy(tmp, [[docInfoList objectAtIndex:(i-1)] docPath]);
                *rindex(tmp, '.')=0;
                NXPrintf(s, "<LINK HREF=\"../%V.\"HTMD_EXT\"/\"HTML_INDEX\"\"
REL=\"Sibling\" TITLE=\"%v\">\n", rindex(tmp, '/')+1, [[docInfoList objectAtIndex:(i-1)]
docTitle]);
            }
        }
    }
// }

// continue for "peer" documents
// for peers, we get a list of doc IDs that are space-separated
i=0; peers = [self peers];
while (peers[i] && (1 == sscanf(peers+i, "%x", &aDocID))) {
    theDocInfo = [docInfoTable valueForKey:docID];
    if (theDocInfo) {
        strcpy(tmp, [theDocInfo docPath]);
        *rindex(tmp, '.')=0;
        NXPrintf(s, "<LINK HREF=\"../%V.\"HTMD_EXT\"/\"HTML_INDEX\"\" REL=\"Peer\"
TITLE=\"%v\">\n", rindex(tmp, '/')+1, [theDocInfo docTitle]);
    }
    while(peers[i] && (peers[i] != ' ')) i++; //skip ahead
    while(peers[i] == ' ') i++; //skip ahead
}

NXPrintf(s, "</HEAD>\n");
return self;
}

- writeHTMLTrailer:(NXStream *)s // optional; user-preference
{
    char tmp[MAXPATHLEN];
    long aDocID;
    id theDocInfo;
    id docInfoList;
    NXAtom peers;
    int i;

    // we should try to check the user's preferences here
    if (![self emitTrailer]) return self;

    // write HR, then parent, peers, then author, date, then
    NXPrintf(s, "\n\n<!-- TRAILER -->\n<HR>\n<DL COMPACT>\n");

    sscanf(tmp, "%x", &aDocID);

```

```

theDocInfo = [docInfoTable valueForKey:aDocID];
if (theDocInfo) {
    strcpy(tmp, [theDocInfo docPath]);
    *rindex(tmp, '.')=0;
    NXPrintf(s, "\n<DT>Go Up (Parent):</DT>\n<DD>[<A
HREF=\"../%V.\"HTMD_EXT\"/\"HTML_INDEX\">%v</A>]\n", rindex(tmp, '/')+1, [theDocInfo
docTitle]);
}

docInfoList = [[NXApp navigator] query:[self docIDStr] field:PARENT];
i = [docInfoList count];
if(i) NXPrintf(s, "\n<DT>For More Details (Children):</DT><DD>\n");;
for (;i;i--) {
    strcpy(tmp, [[docInfoList objectAtIndex:(i-1)] docPath]);
    *rindex(tmp, '.')=0;
    NXPrintf(s, "\n[<A HREF=\"../%V.\"HTMD_EXT\"/\"HTML_INDEX\">%v</A>]\n",
rindex(tmp, '/')+1, [[docInfoList objectAtIndex:(i-1)] docTitle]);
}

// for peers, we get a list of doc IDs that are space-separated
i=0; peers = [self peers];
if(strlen(peers)) NXPrintf(s, "\n<DT>See Also (Peers):</DT><DD>\n");;
while (peers[i] && (1 == sscanf(peers+i, "%x", &aDocID))) {
    theDocInfo = [docInfoTable valueForKey:aDocID];
    if (theDocInfo) {
        strcpy(tmp, [theDocInfo docPath]);
        *rindex(tmp, '.')=0;
        NXPrintf(s, "\n[<A HREF=\"../%V.\"HTMD_EXT\"/\"HTML_INDEX\">%v</A>]\n",
rindex(tmp, '/')+1, [theDocInfo docTitle]);
    }
    while(peers[i] && (peers[i] != ' ')) i++; //skip ahead
    while(peers[i] == ' ') i++; //skip ahead
}

// if ([self parent] != NXUniqueString("")) {
docInfoList = [[NXApp navigator] query:[self parent] field:PARENT];
i = [docInfoList count];
if(i > 1) NXPrintf(s, "\n<DT>See Also (Siblings):</DT><DD>\n");;
for (;i;i--) {
    if ([docInfoList objectAtIndex:(i-1)] != self) {
        strcpy(tmp, [[docInfoList objectAtIndex:(i-1)] docPath]);
        *rindex(tmp, '.')=0;
        NXPrintf(s, "\n[<A HREF=\"../%V.\"HTMD_EXT\"/\"HTML_INDEX \">%v</A>]\n",
rindex(tmp, '/')+1, [[docInfoList objectAtIndex:(i-1)] docTitle]);
    }
}
// }

```

[illegible]