The new Hyperties database format permits files in the database to be organized in an essentially arbitrary way. This permits a user to impose a hierarchical structure (or any other that is suitably mnemonic), using the features of the standard filesystem found in UNIX and MS-DOS, to simplify access to components of the database when a fully-integrated authoring environment is not available, and to "flatten" the structure of a database to optimize use of storage when authoring is fully supported. Also, the new format will allow spreading the component files of a database across multiple devices, allowing larger databases to be used on floppy-based systems.

Several simplifying assumptions have been made, which impose limits on the browser interface. These, however, are not inherent in the file structure, but have been introduced out of considerations of efficiency in implementation. In future versions, some of these restrictions may be relaxed. First, it is assumed that the articles are to be displayed in windows whose size is fixed ahead of browsing time -- the current notion is that construction of a database and formatting of the component articles is to be done ahead of time, in a separate "compilation" phase. Furthermore, articles are to be viewed by paging rather than scrolling, so that formatting produces a collection of page images. On more powerful systems, the formatting could be performed at browsing time for resizable and/or scrolling windows. Finally, a Hyperties database is assumed to be static, i.e. the contents of a database will not change during browsing. With a more powerful database management system to handle file requests and arbitrate access conflicts, dynamic databases should be possible as well.

In general, the files of a Hyperties database fall into two categories, portable (system-independent) files, and non-portable (system-specific) files. Of the portable files, the most basic file type is the **storyboard**, which contains a pure text description of an individual article in the system. Pictures and targets may also be portable, provided that the host browser supports the file format, and can compensate for differences in screen aspect ratio, etc. Non-portable files are the **master index**, which gives the correspondence between identifiers (see below) and system-specific file names, and the **display files**, which describe the images of displayed pages in their formatted form. The latter are non-portable since they are binary files, and make use of system-specific coordinate systems and window sizes.

Since a Hyperties database will consist of several types of component files, there will be a set of standard 3-letter extensions used to identify file types to the system. These extensions will identify the nature of the contents of a file, its system-specificity, and a general version number, to permit some evolution of the internal file formats without "orphaning" existing databases. For example, the initial portable storyboard files will have the extension ".sb0", and a later version of the (non-portable) PS/2 display file may be ".ps3". A further convention adopted is that all database components are referred to by an **identifier**, which is a user-supplied name of arbitrary length, and that components contain their own names, so that they may be identified and collected automatically, without requiring user intervention. These identifiers may contain embedded whitespace, but not leading or trailing whitespace (it will be removed automatically); sequences of embedded whitespace will treated as single, "generic" whitespace characters for purposes of comparison. In addition to the implicit type specified by the extension, the display files contain the images of (typed) internal system objects, whose methods permit these objects to be loaded according to their explicitly-stated types. These images may contain pointers to other object images (on disk), which are simply an encoded filename and offset. These pointers, and the handling of these objects is described in greater detail in the object-system document.

As mentioned above, the **storyboard** files are pure text descriptions of the individual documents (nodes in the hypertext network) of a Hyperties database. These files are broken into five sections, containing the **title** (the identifier of the document), **synonyms** (alternate identifiers by which the document may be referenced), **description** (a brief textual, graphical, etc. piece, usually used to summarize the document), **content** (a lengthy multi-media piece -- the document proper), and **notes** (an optional collection of textual remarks maintained by the author). The content (and possibly the description) may contain **references** to other documents, which are simply the titles or synonyms of those documents, embedded in the text (or connected to a graphic), and marked by the author. Since the storyboard is a pure text document, it uses a special formatting language (see document) to indicate page layout, font types, and to specify the inclusion of graphics.

The **master index** provides the mapping of identifiers to actual file names in a file system. It thus defines the collection of files that

constitute a database. The master index is constructed automatically by the database **compiler**, but is a human-readable text file, so that an author may make use of it to locate database files in the absence of an authoring environment, which would automatically provide reference by identifier. The format of this file is as follows:

1) a header line, indicating that document records follow, followed by a blank line

2) a collection of document records, each consisting of:
   - a title, surrounded by quotes, with "" for embedded quotes
   - a pathname, relative to the database directory (or common ancestor of all database directories), no extension
   - (on following lines) synonyms, formatted like the title, without pathname
   - A type identifier, that tells Ties how to handle this document.

3) a header line, indicating that picture records follow, preceded and followed by blank lines

4) a collection of picture records, each consisting of:
   - identifier (like title)
   - pathname with extension
   - byte offset (since one file may contain several pictures)

5) a header line, indicating that target records follow, preceded and followed by blank lines

6) a collection of target records, identical to picture records

7) a header line, indicating that the file map follows, preceded and followed by blank lines

The **display files** are created when individual storyboards are formatted for a specific system and configuration. These files, essentially, contain nothing but formatting information – the actual text of a document comes from the associated storyboard. In this way, formatting on-the-fly is eliminated, allowing a document to be displayed very rapidly, but without having to keep two copies of every document (the formatted and unformatted versions). As mentioned above, the display files are binary images of the internal objects used to represent components to be

Support lazy compilation, (checks for already compiled files, if not there, then compile them) and on the fly compilation. [always compile, Compiler generates no output files, JUST a view.)

displayed. Each record of this file contains a type ("class") identifier that is used by the object system to invoke the appropriate functions to read (or skip) the record. In this way, the format of the display file need not remain fixed as the capabilities of the browser are extended; such extensions are incorporated by creating new classes of objects, while the old classes continue to be supported. These display files are organized as follows:

and a length (byte count) [so we don't actually have to implement a class to skip over it]

Display file:

<type DocumentHeader> <header info (title, no. of pages, etc)>
<type Description> <description data object>*
[<type Page> <page data object>*]*

Description and page data objects:

<type Text> <font info (type, size, etc)> <# of strings>
<x, y, length, offset in storyboard>* |
<type TextTarget> <font info> <# of strings>
<x, y, length, offset in storyboard>* |
<type Picture> <x, y, length, offset in storyboard>* |
<type ~~Picture~~Target> <x, y, length, offset in storyboard (target id), length, offset in storyboard (reference id)>

Display routine optimizes out unnecessary font changes when creating display list.

reference id? (or is it just the string itself?)

A Hyperties database will also contain a number of **auxiliary files**, which allow images (and associated targets), generated by various graphics editors, to be included. These exist simply to provide an identifier, and description of the format, for such images, without having to modify the image files themselves (so they remain editable). These auxiliary files, which are also human-readable, consist of an arbitrary number of records, each of the following form:

- type (identifies picture or target, image file format, etc)
- identifier
- name of picture or target data file

The process of **compiling** a Hyperties database consists of several phases. First, the given directory or directories which contain the database are scanned for files with the appropriate extensions, and the three parts of the index (storyboards, pictures, and targets) are created by examining these files for their identifiers. This phase also involves some

checks for name conflicts. Then, the individual storyboards are processed by the formatter to create the display files and extract references to other documents in the database. At this point, all references are checked against the index – if any identifiers are missing from the index, the user is requested to supply their location if possible. The formatting process generates the display files, using information contained in the current environment description. Last, any uncorrectable errors in the database are signalled to the user, and various cross-reference tables may be produced, if requested. Ultimately, the compiler will support partial compilation, for case where only a small number of documents changed. This is a necessity for large databases, where the cost of recompiling an entire database could be substantial.

The database structure and compilation process described above allow for a variety of authoring aids, short of a full authoring environment, to be constructed simply. One possibility is a simple file selection utility, which would allow the author to select a document or picture by its identifier (a long, mnemonic name), and, using information contained in the master index, automatically invoke a word or graphics processor on the appropriate file(s), and recompile the database as changes are made. This could very likely be written using shell scripts or batch files in a UNIX or MS-DOS environment. Another tool of great utility would be an improved facility for selecting targets in graphical images.

- overall organization of h.t. on SUN
    - display (SunView version, later X or NeWS)
    - interface driver (control of interaction).
    - database manager (simple indexer)
    - document manager (maintains piles and access path)
- object-oriented C programming
    - coarse-grained object structure
    - no inheritance
    - each module defines a single class
        - metaclass (class methods)
            - static OBJECT, class is itself, no external name
            - struct of pointers to class methods (functions)
        - class (instance methods)
            - static OBJECT, class is its metaclass
            - struct of pointers to instance methods (functions)
            - externally accessable thru OBJECT classname
            - receives create, init, etc. messages
        - instance variables (object struct)
        - class variables (static to module)
        - set of static functions for class and instance methods
            - each message type has a single return type
                (OBJECT, int, double, void, pointer, etc.)
            - provide some global methods (Error, TypeOf, etc.)
            - all take at least one param (first), self.
    - provides opaque pointer type OBJECT
        - all objects have a class
        - all classes are objects, have type id numbers and names
    - message syntax (expression with value of return type)
        - (SEND (*object*, *message*) WITH *param1*, ... , *paramN*)
        - this produces:
        (* *object* ->class.*message*) (*object*, *params1*, ... , *paramN*)