This document describes data structures, designs and concepts which are the proprietary intellectual property of Xanadu Operating Company, Inc. The contents of this document are not for distribution or release in whole or in part to any other party without the express permission of Xanadu Operating Company, Inc. All portions of this document are to be considered trade secrets of Xanadu Operating Company, Inc. including the fact that some previously published data structures may fall into the classification of "enfilades".

WARNING!

He who transgresses against the propriety of the Information contained herein shall be Cursed! Woe unto all who reveal the Secrets contained herein for they shall be Hunted unto the Ends of the Universe. They shall be afflicted unto the Tenth Generation with Lawyers. Their Corporate Bodies shall be Broken and cast into the Pit. Their Corporate Veil shall be Pierced, and Liability shall attach to the Malefactors in personem. They shall suffer Ulcers and Migraines and Agonies Unimagined. Yea, Verily, for such shall come to pass against all who would Dare to Test the Powers of Xanadu unto their Doom.

Xanadu Hypertext Documents
Table of Contents

I. Xanadu Operating Company / Xanadu System Proposal
II. System Architecture
III. Enfilade Theory
IV. Data Structures
V. System Implementation
VI. Xanadu Hypertext Virtuality
VII. Future Directions for Development
VIII. Analysis of the Memory Usage and Computational Performance
IX. Frontend/Backend Interface
X. Annotations to the Backend Source Code
XI. Vitae of Key Personel
XII. Glossary of Terms
XIII. Outline of Development Plan Tasks

Proposal for the Implementation by Xanadu Operating Company of a Full-scale Semi-distributed Multi-user Hypertext System

Xanadu Operating Company (XOC) is developing a hypertext information storage management system called "Xanadu". Hypertext is text or data which exhibits patterns of structure or interconnectedness which are not necessarily very regular and which may change over time. Project Xanadu, the informal group that became XOC, has developed a family of data structures which in combination appear to meet the requirements of such a system. A working prototype exists implementing the rudiments of the design. Since its computational complexity and storage overhead are essentially logarithmic with the volume of stored information, it should be able to efficiently handle very large amounts of material.

XOC intends to develop hypertext system software beyond the present prototype stage, producing a fully operational and usable system. Specifically, we propose to transport the current software to the Interlisp environment, extend the design, and complete its implementation. The system will then be tuned and optimized to increase its speed and compactness. The end product of this effort will be a complete "semi-distributed" multi-user Xanadu "backend" integrated with the Interlisp environment and able to serve as an Ethernet-based resource for a wide variety of applications. Details of the development plan, including a complete schedule and budget, are included in the attached documents.

We believe the benefits of our system will be longer-term than is typical with commercial ventures. An instantiation of these ideas in a working database manager will be of immediate benefit to researchers in many fields. SDF's clients appear most likely to be able to perceive and utilize the potential in these ideas and their instantiation. Not only do these people have a present need for this sort of tool, but they also will provide an excellent community of critics to stimulate the refinement and generalization of both the design and the implementation.

The Xanadu Hypertext System Architecture

The Xanadu Hypertext System manages the storage, retrieval and manipulation of character strings and orgls. An orgl is a structure which represents the organization of a collection of address spaces (called virtual spaces or V-spaces for short), each containing an editable stream of atoms. This stream is referred to as the virtual stream or the variant stream (or V-stream for short).

An atom is the primitive element that the Xanadu System deals with. There are two types of atoms in the present system: orgls, representing structural and organizational information, and characters (8-bit bytes), representing actual data. Other types of atoms are conceivable, such as videodisk frames or other kinds of (non-orgl) organizational structures, but these are not present in the current design.

Any atom contained in the system may be referenced by specifying its virtual stream address. This is recursively defined as the virtual stream address of the orgl containing the atom, combined with the number of the V-space which holds the atom within that orgl and the position of the atom itself within that V-space. Orgls that are not contained within other orgls are addressed by a special sort of V-stream address called an invariant orgl identifier, terminating the recursion. Thus, a virtual stream address contains a variant part and an invariant part which are syntactically separable.

The contents of the V-spaces within an orgl may be edited. This in turn means that the V-stream addresses of atoms within an orgl may change, as implied by the adjective "variant". The contents of an orgl may be added to or deleted from at any point in a V-space. In addition, sections of a V-space may be rearranged (i.e., a section may be moved or two sections transposed). These operations -- insert, delete and rearrange -- can cause the position and relative ordering, the V-stream order, of atoms to shift, thus altering those atoms' V-stream addresses as well.

Atoms are stored internally in an invariant stream (or I-stream for short). They appear, and are addressed, in I-stream order. As the adjective "invariant" suggests, the I-stream address of an atom never changes. The function performed by an orgl is the mapping back and forth between I-stream addresses and V-stream addresses. When the contents of a V-space are edited, the orgl mapping that V-space is changed, and thus so are the virtual positions of the atoms.

The I-stream addresses of atoms are not generally visible externally. The only exceptions are invariant orgl identifiers which are in fact the I-stream addresses of "top level" orgls.

The above discussion refers to mappings to and from particular I-stream and V-stream addresses. In practice, we work with spans rather than point addresses. A span is an abbreviated way of referring to a group of contiguous addresses. A span consists of a starting address together with a length. A V-span is a span of V-stream addresses, and an I-span is similarly a span on the I-stream. An orgl maps from one V-span to a set of I-spans, and from an I-span to a set of sets of V-spans.

The actual atoms located in a particular I-span may be found using another structure called the grandmap. The grandmap is a tree of pointers to all the atoms currently stored in the system, indexed by I-stream address. The grandmap provides a mapping between I-stream addresses and the locations of the actual underlying physical storage used to hold the atoms.

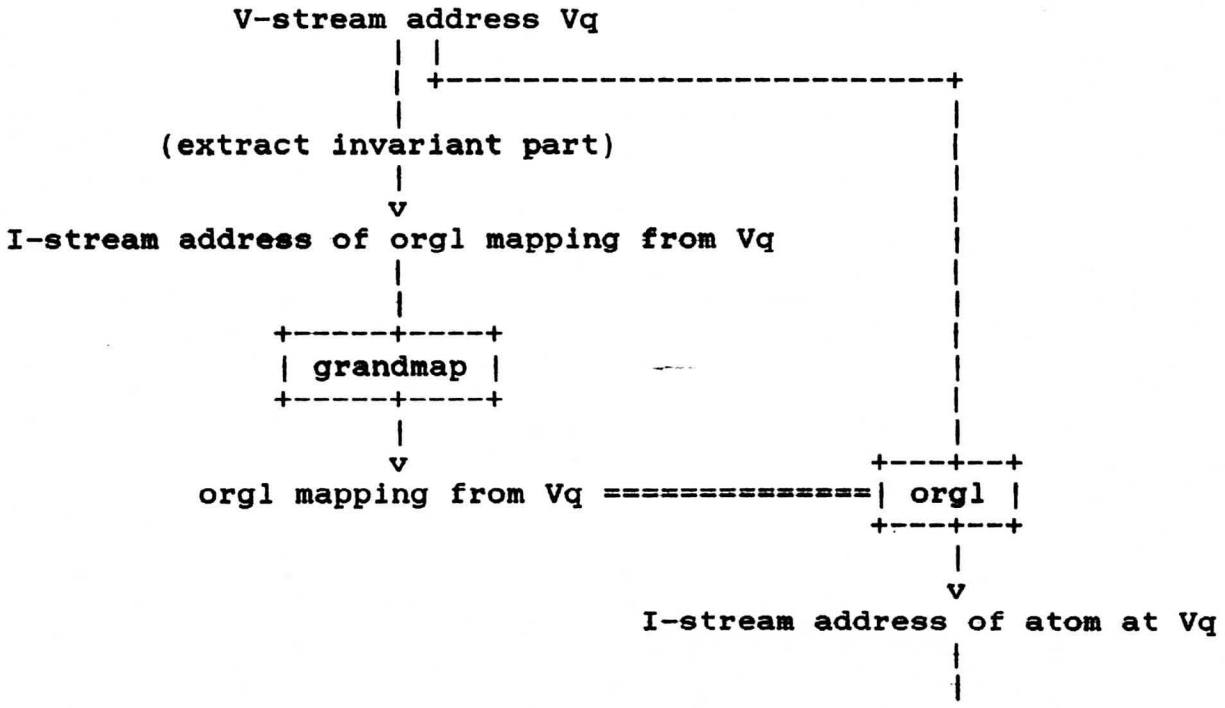
There are thus three levels of addressing used in this system:

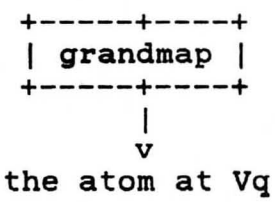
V-stream addresses identify atoms to the outside world. The V-stream address of an atom may change and, as will be explained below, an atom may have more than one V-stream address.

I-stream addresses identify atoms internally in a consistent and implementation independent fashion. The I-stream address of an atom in the Xanadu system is analogous to the accession number of a document in a library. An atom has but one I-stream address and this address never changes.

Physical storage addresses identify the locations of the actual bits and bytes of atoms themselves. An atom's physical address is both variable and highly implementation dependent. For example, it may change due to reorganization of the underlying storage for purposes of efficiency or convenience, or due to changes in the types of storage devices used.

To retrieve the atom located at a particular V-stream address Vq, the invariant part of Vq's V-stream address specification is extracted. This will be an invariant orgl identifier for the orgl which maps Vq to some I-stream address. Since this invariant orgl identifier is itself an I-stream address, it may be (and is) used as an index into the grandmap to retrieve the relevant orgl. Vq (the full V-stream address, not just the variant part) is then mapped through this orgl to its corresponding I-stream address. This second I-stream address in turn is used for a second lookup in the grandmap to acquire the atom which Vq addresses. The following diagram illustrates the data flow:





An important consequence of the V-stream to I-stream mapping is that several V-stream addresses may all map to the same I-stream address. This means that an orgl may contain multiple virtual copies of a given set of material. Although the I-stream address of an atom is related to the orgl in which it originally was inserted, the V-stream address(es) of that atom are under no such constraint. Therefore, virtual copies of material originating in one orgl may appear in other orgls.

Another consequence of this structure is that multiple orgls may be used to represent alternate V-to-I mappings for the same set of atoms, yielding alternative versions of a given collection of material. In practice, the data structures used to realize such orgls can share their underlying components in those places where the V-to-I mappings are similar and need only differ in their structure where the versions are actually different. (See the accompanying paper on the implementation of the Xanadu internal data structures for a detailed description of how this mechanism works).

The orgls which represent multiple versions of the same family of material are collectively called a phylum. All of the orgls in a phylum are "descended" from a single original orgl, in the sense that they were created by applying some edit operations to that orgl or to one of its later descendents.

Xanadu provides a facility called historical trace. The sequence of edit operations that have been applied to the orgls of a phylum can be seen to form an historical trace tree. This tree branches wherever a different version was created. Such versions result when two or more processes modify the same orgl through different berts or when a process backs an orgl up to an earlier state and then makes edits.

Since each of the editing primitives which the system supports is a reversible operation upon an orgl, the state of an orgl at any point in its history conceivably might be obtained by inverting each of the edits in an edit log. However, a data structure is constructed which represents the edit history of a phylum at varying levels of detail. This data structure is a tree which at the bottom level represents individual edit operations and at the higher levels represents compositions of these lower level edits -- "superedits" that are single transformations that represent the end result of a series of more primitive operations. This tree is indexed by position in the phylum's historical trace tree (note that there are two trees here: the historical trace tree, representing the sequence of operations forming the phylum's history, and the data structure erected over this tree, representing the operations themselves). The end result of a retrieval operation on this data structure is not a series of edit operations but the actual V-to-I mapping that existed at the indicated point in the phylum's history. The details of how this is accomplished are given in the accompanying paper which describes the data structures themselves.

The system can retrieve any atom stored within it, given the atom's

V-stream address. In the case of a character atom, the retrieval operation returns the character itself. In the case of an orgl atom, an entity called a bert is returned. A bert is an identifier for an orgl pointing to a current access, as opposed to an established V-stream address. It grants the process to which the bert is given exclusive access to a particular V-to-I mapping.

A process, in the Xanadu System, means a particular external connection to the system which may request the retrieval, creation and editing of orgls and characters. An arbitrary (i.e., implementation dependent) number of processes may access the system concurrently.

Separate processes which request the retrieval of the same orgl at the same time are each given different berts which refer to the orgl. Associated with each orgl is a count of the number of berts which currently refer to it. If one of these processes then makes an edit change to the orgl, a new orgl will be created. The process's bert will be made to refer to the new orgl and the old orgl's reference count will be decremented. By this means, the other processes will not "see" the change, and their berts will still refer to the same V to I mapping as previously. Any information about the orgl's state which the other processes might have been keeping externally will not be invalidated by the one process's edit operation.

A structure called the spanmap implements two mappings. The first of these is from the I-stream addresses of atoms in general to the I-stream addresses of orgls. The second mapping is from the same original I-stream addresses to berts. The spanmap is designed to answer querys about which orgls reference which I-spans. It quickly identifies the orgl or orgls that map some V-stream address(es) onto a particular I-stream address. This is the inverse of the set of mappings implemented by the full collection of orgls stored in the system.

The spanmap, as its name suggests, is fundamentally designed to deal with spans. It enables the system to answer queries about which orgls refer to particular pieces of material, given the V-stream addresses of the atoms of interest. The general form of such a query takes the form of the question, "what are the V-stream addresses of all the orgls which refer to this particular V-span?" The V-span of interest is mapped to a set of I-spans (call this set Q, for "query), using the procedure described above in the discussion of the operation of orgls. This I-span set, Q, is then used to initiate a lookup in the spanmap which results in a set of one or more I-stream addresses corresponding to the orgls which reference Q (along with berts identifying orgls that reference Q but may not yet have I-stream addresses). These I-stream addresses are then looked up in the grandmap, yielding the orgls themselves. These orgls are in turn used to map Q to a set of V-spans (keep in mind that the mapping implemented by an orgl is bidirectional) which are then returned as the answer to the process which initiated the query.

Since the number of orgls which might refer to a particular I-span is potentially very large, the spanmap enables restricted retrievals. A retrieval may be restricted to return only orgls from a particular set, for example those which reference a particular I-span in addition to the one of interest. This is important, among other reasons, because queries are generally expressed in terms of V-spans, and a single V-span may map to a number of I-spans. If one wishes to determine the set of orgls which reference a particular V-span, what is desired is the intersection of the sets of orgls

that reference the I-spans that the V-span of interest maps to.

The spanmap also enables queries that may be expressed in terms of the overlap of spans, rather than simple reference. Thus one may ask which orgls contain I-spans that begin inside a particular span and end outside of it, for example. The reader is again referred to the companion paper on the implementation of the various data structures.

In summary then, the system consists of four primary components:

1) orgls, which map back and forth between V-stream and I-stream addresses,

2) the grandmap, which maps from I-stream addresses to the physical addresses of atoms (characters and orgls) themselves,

3) the spanmap, which maps from the I-stream addresses of atoms to the I-stream addresses of orgls which reference those atoms, and

4) an historical trace, which enables the determination of the contents of an orgl at any point in its history or the history of any of its other versions.

Enfilade Theory

The underlying data abstraction of the Xanadu System is the enfilade. The grandmap, spanmap and orgls are all implemented using different types of enfilades. We shall first discuss enfilades generally and then show how the theory is adapted to specific implementation.

DEFINITIONS

An enfilade is a data structure in which the positions of data items in index space (i.e., the retrieval keys associated with those data items) are stored indirectly, as local positions relative to the data items' neighborhoods in the data structure, rather than being stored directly, as absolute positions.

Enfilades have traditionally been implemented as trees. Each node of the tree is called a crum (in order to disambiguate the term "node" -- see footnote 1). Each crum contains, either explicitly or implicitly, two components of indexing information, called the wid and the disp. In addition, a crum may contain other structural information, such as pointers to descendent or sibling crums.

A disp represents the relative offset, in index space, of its crum from its crum's parent. The "sum" of a crum's disp with those of all of its ancestors determines the crum's absolute position in index space. A change in a crum's disp therefore causes a corresponding change not only in the crum's absolute position but in those of all of its descendents.

A wid represents the extent, in index space, of its crum's collective descendants, relative to its crum's position (which is in turn derived from its crum's disp). A change in a crum's wid may result if the wid or disp of one of the crum's children changes.

An enfilade's disps are interpreted in a top-down fashion, telling the relationship of crums to their ancestors, while the wids are interpreted from the bottom-up, indicating the relationship of crums to their descendents.

A group of sibling crums, all descended from the same parent crum, is collectively referred to as a loaf or crum-block. The single crum which forms the root of the tree and from which all other crums are descended is called the fulcrum. The disp of the fulcrum is offset relative to the origin of the index space. The crums which form the leaves of the tree are called bottom crums. The tree is maintained at a constant depth, so all bottom are the same number of levels below the fulcrum. Bottom crums may contain actual data in addition to structural and indexing information and therefore may have a different structure or form than their ancestors do. Non-bottom crums are referred to as upper crums. When ennumerating the levels of an enfilade, it is conventional to number from the bottom up so that the level number of a crum will not change as the enfilade grows or shrinks (levels are added or deleted at the top).

An index space may be multi-dimensional, and not all dimensions need participate in enfiladic operations, as described below.

An enfilade is similar in many ways to a conventional B-tree. This is especially so when considering the set of primitive operations that are

defined. However, an enfilade is not a B-tree. Enfilades possess two properties that distinguish them from B-trees (properties that were, in fact, the motivation for the invention of the first enfilades). These properties are rearrangability and the capacity for sub-tree sharing. These derive from the nature of wids and disps as local abstractions independent of the overall frame of reference of the full data structure.

While enfilades have traditionally been implemented as trees, we do not feel that this is essential. Generalization of the theory of enfilades to other types of data structures, for example hash tables, has been considered but not yet examined in depth.

OPERATIONS

The fundamental operations on an enfilade are retrieve, rearrange and append. These are augmented by the useful, though not strictly necessary, operations insert and delete. All of these are supported by the "housekeeping" operations cut, recombine, level push and level pop.

The retrieve operation obtains a data item associated with a particular location in index space. Such a data item may be stored in an enfilade either directly, by actually storing it in the bottom crum associated with the desired position in index space, or indirectly, as a function of the wids and disps of the crums traversed while descending the tree to that bottom crum. The latter alternative is fairly unusual and deserves elaboration. For example, enfilades in index spaces with multiple independent dimensions could store data by indexing with some dimensions and not others and then return the positions of bottom crums along the other dimensions. A single enfilade may contain several collections of data at once, some stored one way and some another.

The general algorithm for retrieve is:

```
retrieve (indexSpacePosition)
  result <-- recursiveRetrieve (indexSpacePosition, fulcrum, .0.)
end retrieve

recursiveRetrieve (index, aCrum, cumulativeIndex)
  if (index .==. cumulativeIndex)
    result <-- data (aCrum)
  else
    for each child of aCrum
      if (disp(child) .<=. index) and (index .<. (disp(child) .+. wid(child)))
        result <-- recursiveRetrieve (index, child, cumulativeIndex .+.
                                                              disp(child))
      end if
    end for
  end if
end recursiveRetrieve
```

where disp(crum) extracts a crum's disp, wid(crum) extracts its wid, and data(crum) extracts the datum from a bottom crum. The symbols .==., .<. and .<=. represent comparison operators in index space. The symbol .+. represents "addition" in index space. The symbol .0. represents the origin of the index space. If less than the full number of possible dimensions is being used for indexing, the index space operators must be specified to take

this into account. The symbol <-- is an assignment operator. result is assigned to to return a value. The control constructs are what they appear to b

Note that this algorithm retrieves exactly one data item stored directly in a bottom crum. If data are stored indirectly in the wids and disps, as described above, the algorithm must be modified accordingly. To implement the case described previously (where some dimensions are used for indexing and others for indirect data storage), the operators .==., .<., and .<=. should be defined only to compare along the indexing dimensions, while .+. should be defined on all dimensions. The result returned should not be the extraction of data from the bottom crum but the extraction of the data dimension components of cumulativeIndex.

Also note that it is possible for a single index space position to map to more than one data item, although this algorithm would only return the last of these. The possibility of such multiple hit retrievals is a general property of enfilades, although specific types of enfilades may by their nature exclude it (multiple hit retrievals may occur, for example, in the case of a multi-dimensional index space when data are retrieved using indices with less than the full number of dimensions). The obvious generalization of collecting multiple data items into a set was omitted for the sake of clarity.

This algorithm also does not take into account the possibility that the desired data item might not be present at all. Once again, the generalization of collecting the results in a set would correct this (the result in such a case would be an empty set) and the omission is for clarity.

In cases where multiple hits are excluded, retrieval time is logarithmic with the number of data items stored. Where multiple hits are permitted, non-logarithmic elements are introduced and the analysis is not so straightforward, but depends upon the specific nature of the enfilade in question. In the case of multi-dimensional index spaces, retrieval times depend on the splitting and regrouping algorithms used to balance the tree. There are tradeoffs that depend on the number of dimensions that are typically to be used to retrieve with and the performance in atypical retrievals. If the enfilade is totally optimized along one dimension, retrievals will be of logarithmic order along that dimension and linear along the others. If it is optimized along D dimensions collectively and retrievals are performed along K of those dimensions, the retrieval time will be on the order of N**((D-K)/D)*log(N), K<=D, where N is the number of data items stored in the enfilade.

The rearrange operation alters the index space positions of clusters of data items by selective alteration or transposition of disps. Rearrangability is one of the two essential properties of enfilades which distinguish them from other sorts of data structures (the other, sub-tree sharability, is discussed below). Rearrange changes the relative ordering of crums in index space.

Rearrange operations are specified in terms of cuts. There are two "flavors" of rearrange, called three-cut rearrange and four-cut rearrange. A cut delineates a boundary for the rearrange operation. A cut is a separation between two regions of interest, defined by a position in index space, C, such that there exists a pair of sibling crums (i.e., crums descended from the same parent), A1 and A2, such that:

```
indexSpacePosition(A1) .<=. C .<. indexSpacePosition(A2)
```

```
                    and
(indexSpacePosition(A1) .+. disp(A1)) .<= C
```

and, for any crum Q in the enfilade at a level lower than that of A1 and A2:

```
indexSpacePosition(Q) .<=. C implies that either
        indexSpacePosition(Q) .<. indexSpacePosition(A1)
                or
        Q is a descendant of A1

and

C .<. indexSpacePosition(Q) implies that
        indexSpacePosition(A2) .<=. indexspacePosition(Q)
```

where, if the index space is multi-dimensional, the comparison operations are limited to the dimensions along which the cut is being performed (cuts are often made along less than the full number of dimensions).

Cuts are generally used in groups (in the case of rearrange, in groups of three or four). When multiple cuts are used together they are constrained to propagate upward until all cuts reach a common ancestor. In other words, if cut C1 is bounded by crums A1 and A2, as described above, and cut C2 is similarly bounded by crums A3 and A4, then crums A1, A2, A3 and A4 should all have the same parent.

A three-cut rearrange performed with cuts C1, C2 and C3 moves the material between C2 and C3 to the position defined by C1 (or, equivalently, moves the material between C1 and C2 to the position define by C3). This is accomplished at the level of the sibling crums at the top of the three cuts by adjusting some of the crums' disps, as follows, where P is the parent to all of the crums at the top of the cuts:

```
for each crum that is a child of P
        pos <-- indexSpacePosition(crum)
        if (C1 .<. pos) and (pos .<=. C2)
                disp(crum) <-- disp(crum) .+. (C3 .-. C2)
        else if (C2 .<. pos) and (pos .<= C3)
                disp(crum) <-- disp(crum) .-. (C2 .-. C1)
        end if
end for
```

where the symbol .-. represents "subtraction" in index space.

A four-cut rearrange performed with cuts C1, C2, C3 and C4 transposes the material between cuts C1 and C2 and the material between cuts C3 and C4. As with the three-cut rearrange, this is performed by manipulating the disps of the crums at the top of the cuts:

```
for each crum that is a child of P
        pos <-- indexSpacePosition(crum)
        if (C1 .<. pos) and (pos .<=. C2)
                disp(crum) <-- disp(crum) .+. (C4 .-. C2)
        else if (C2 .<. pos) and (pos .<=. C3)
                disp(crum) <-- disp(crum) .-. (C2 .-. C1) .+. (C4 .-. C3)
        else if (C3 .<. pos) and (pos .<=. C4)
```

```
                        disp(crum) <-- disp(crum) .-. (C3 .-. C1)
                end if
        end for
```

It can be seen that the three-cut rearrange is equivalent to a four-cut rearrange in which two adjacent cuts are identical.

The cut operation itself is accomplished by splitting crums which straddle the cut location. The two new crums correspond to the two sides of the cut. The children of the old crum are assigned to the new crums according to which side of the cut they fall on -- any children which themselves straddle the cut are split using the same procedure recursively. Cuts are usually made in groups, with the cutting process terminating when a single crum spans all of the cut location (this crum corresponds to the crum P in the algorithms above). The following is the algorithm for cut:

```
cut (cutSet)
        recursiveCut (cutSet, fulcrum)
end cut

recursiveCut (cutSet, parentCrum)
        dontDiveDeeperFlag <-- TRUE
        for each child of parentCrum
                if (disp(child) .<. firstCut(cutSet)) and (lastCut(cutSet)
                                                .<=. (disp(child) .+. wid(child)))
                        dontDiveDeeperFlag <-- FALSE
                        for each cut in cutSet
                                cut <-- cut .-. disp(child)
                        end for
                        recursiveCut (cutSet, child)
                end if
        end for
        if (dontDiveDeeperFlag)
                chopUp (cutSet, parentCrum)
        end if
end recursiveCut

chopUp (cutSet, parentCrum)
        for each cut in cutSet
                for each child of parentCrum
                        if (disp(child) .<. cut) and (cut .<=. (disp(child)
                                                        .+. wid(child)))
                                newChildSet <-- split(cut, child)
                                disown(parentCrum, child)
                                adopt(parentCrum, leftChild(newChildSet))
                                adopt(parentCrum, rightChild(newChildSet))
                                break out of inner loop
                        end if
                end for
        end for
end chopUp

split (cut, crum)
        leftCrum <-- createNewCrum ()
        rightCrum <-- createNewCrum ()
        disp(leftCrum) <-- disp(crum)
```

```
        wid(leftCrum) <-- cut .-. disp(crum)
        disp(rightCrum) <-- cut
        wid(rightCrum) <-- wid(crum) .+. disp(crum) .-. cut
        for each child of crum
                if ((disp(child) .+. wid(child)) .<. cut)
                        adopt (leftCrum, child)
                else if (cut .<=. disp(child))
                        adopt (rightCrum, child)
                else
                        newChildSet <-- split(cut .-. disp(child), child)
                        adopt (leftCrum, leftChild(newChildSet))
                        adopt (rightCrum, rightChild(newChildSet))
                end if
        end for
        result <-- makeChildSet(leftCrum, rightCrum)
end split
```

where makeChildSet(leftCrum, rightCrum) takes two crums and returns them in some sort of ordered collection, leftChild(childSet) returns the first child in such a collection, and rightChild(childSet) returns the other child; adopt(parent, child) adds the crum child to the set of children of parent and disown(parent, child) removes it (discarding child); createNewCrum() creates a new, uninitialized crum; and firstCut(cutSet) returns the first (in index space) cut in a set of cuts, and lastCut(cutSet) similarly return the last one.

The append operation adds new elements to the data structure by extending the range of index space covered by it and associating the new elements with these extended index space positions.

The general algorithm to append a single new element to an enfilade is:

```
append (newThing, beyond, where)
        potentialNewCrum <-- recursiveAppend (newThing, fulcrum, beyond, where)
        if (notNull(potentialNewCrum))
                levelPush (potentialNewCrum)
        end if
end append

recursiveAppend (newThing, parent, beyond, where)
    if (where .==. .0.)
        newCrum <-- createNewBottomCrum ()
        data(newCrum) <-- newThing
        wid(newCrum) <-- naturalWid(newThing)
        disp(newCrum) <-- disp(parent) .+. beyond
        result <-- newCrum
    else
        for each child of parent
            if (disp(child) .<=. where) and (where .<. (disp(child) .+.
                                                        wid(child)))
                potentialNewCrum <-- recursiveAppend(newThing, child, beyond,
                                                     where .-. disp(child))
                break
            end if
        end for
        if (notNull(potentialNewCrum))
```

```
            if (numberOfChildren(parent) >= MaximumNumberOfCrumsInALoaf)
                newCrum <-- createNewCrum ()
                disp(newCrum) <-- disp(potentialNewCrum)
                disp(potentialNewCrum) <-- .0.
                wid(newCrum) <-- wid(potentialNewCrum)
                result <-- newCrum
            else
                wid(parent) <-- enwidify(children(parent), potentialNewCrum)
                adopt(parent, potentialNewCrum)
                result <-- NULL
            end if
        else
            result <-- NULL
        end if
    end if
end recursiveAppend
```

where naturalWid(dataItem) is a function that determines the wid of a bottom crum associated with a particular data item and enwidify(crum1, crum2, ...) is the widdative function which computes the wid of a parent crum from the wids and disps of its children. The widdative function is one of the fundamental operators that defines an enfilade, along with .+. and .-.. The location appended to is represented by the two arguments where and beyond which indicate the position in the enfilade to which the new data element is to be appended and the distance beyond that position that will define the new data element's own position. The value MaximumNumberOfCrumsInALoaf sets a limit to the amount of "fanout" at each level of the tree.

Enfiladic trees are generally balanced by maintaining the requirement that the number of children of any one crum (i.e., the number of crums in a loaf) may not exceed a given threshold. In practice, since the structure of bottom crums and upper crums may differ, it is often the case that this threshold will differ between bottom loaves and upper loaves. The actual values chosen for these thresholds depend upon the actual enfilade in question, and are typically selected to optimize retrieval speed, disk space efficiency, or some other empirically determined, implementation dependent criteria.

The level push operation adds an additional level to the tree when an append or insert operation causes the number of children of the fulcrum to grow too large. It simply creates a new fulcrum from which is descended the old one. The algorithm is:

```
levelPush (newCrum)
        newFulcrum <-- createNewCrum ()
        disp(newFulcrum) <-- .0.
        wid(newFulcrum) <-- enwidify(fulcrum, newCrum)
        adopt (newFulcrum, fulcrum)
        adopt (newFulcrum, newCrum)
        fulcrum <-- newFulcrum
end levelPush
```

where the argument newCrum represents the new sibling to the old fulcrum from which is descended the branch of the tree which caused the old fulcrum to overflow.

The insert operation adds a new data element at a random position in the

enfilade. Insert is not a strictly necessary operation, since it is functionally equivalent to an append operation followed by a rearrange operation. The append adds the new item to the data structure and then the rearrange relocates it to the desired location. In practice, insert is often implemented as a separate operation, for reasons of efficiency or convenience. An insertion is accomplished by making a cut at the desired insertion point, extending this cut upwards to a crum whose wid is large enough to encompass the data to be inserted, and then plugging the new material in. The disps of crums .>. the new material at this level are then incremented accordingly. The algorithm to accomplish this is left as an exercise for the reader in order not to overly extend this paper.

The delete operation removes things from an enfilade. Delete, like insert, is also not strictly necessary, since undesired material can be relocated to any arbitrary "purgatory" by the rearrange operation. In actual use, however, it often desirable to actually delete things in order to be able to reclaim the storage that they occupy. The delete operation is quite simple: two cuts are made on the boundaries of the unwanted region of index space and propagated up to a common ancestor. The child crums of this ancestor which lie between the cuts are then disowned and the disps of greater siblings reduced accordingly. The disowned crums and all of their descendants may then be deallocated or left for garbage collection.

Deletes and rearranges can result in an enfilade that is a badly fragmented, unbalanced tree and in which many crums have fewer than the optimum number of children. This in turn can result in a tree which has more levels than necessary, with a potentially adverse affect upon performance. The recombine operation is used to tidy things up by merging sibling crums. The algorithm to merge two siblings is:

```
primitiveRecombine (parent, sibling1, sibling2)
        newCrum <-- createNewCrum ()
        disp(newCrum) <-- disp(sibling1)
        for each child of sibling1
                disown(sibling1, child)
                adopt(newCrum, child)
        end for
        dispCorrection <-- disp(sibling2) .-. disp(sibling1)
        for each child of sibling2
                disown(sibling2, child)
                disp(child) <-- disp(child) .+. dispCorrection
                adopt(newCrum, child)
        end for
        wid(newCrum) <-- enwidify(children(newCrum))
        disown(parent, sibling1)
        disown(parent, sibling2)
        adopt(parent, newCrum)
end primitiveRecombine
```

The term recombine is commonly used to refer to the process of climbing around the enfiladic tree and selectively applying the above operation in order to perform some general housecleaning. Such a process may also include cut operations to split up recombined crums which have too many children, perhaps interleaving cuts with primitiveRecombines in order to "shuffle" crums around in the tree. The methods for doing this are heuristic rather than algorithmic, and depend both on the nature of the particular enfilade in question and the

data with which it is being used. Certain applications may not, in fact, require any recombines to be performed at all.

A recombine may also invoke a level pop operation to remove an excess level from the tree. This can be performed when the fulcrum has but a single child. The algorithm is simply:

```
levelPop ()
        newFulcrum <-- theOneChildOf(fulcrum)
        disp(newFulcrum) <-- disp(newFulcrum) .+. disp(fulcrum)
        disown(fulcrum, newFulcrum)
        fulcrum <-- newFulcrum
end levelPop
```

where theOneChildOf(fulcrum) extracts the fulcrum's (presumably) only child.

OBSERVATIONS

The use of wids and disps means that each crum in an enfilade is located relative to its parent rather than to any absolute coordinate space. This in turn means that enfilades may engage in sub-tree sharing. Multiple crums on a given level of one or more enfilades may point to a single lower crum as one of their children. This has the effect of making virtual copies of the sub-tree represented by that crum and all of its descendents. The index space position of the bottom crums of such a sub-tree depends upon the particular parent crum through which they are accessed. Sub-tree sharability and rearrangability, both the result of the localizing action of wids and disps, are the two properties which most significantly distinguish enfilades from other sorts of data structures.

One of the problems with multi-dimensional data is that, unlike one dimensional data, there is, in general, no single well-defined ordering of the data. K-ary trees (insert reference here) solve this problem using projection onto a single dimension. Enfilades use the locality of wids and disps to eliminate the need for ordering.

One of the many useful applications of sub-tree sharing is versioning -- the creation enfilades which represent alternate organizations of a given body of data. Often, alternate versions of some set of material will have significant portions in common (insert reference on Reps' subtree sharing here) (this, in fact, may be what we mean when we say two things are "versions" of each other, rather than saying that they are separate things). Common portions of two data sets can be represented by a single enfiladic sub-tree. Separate data structure is only required where they actually differ. If the differences between two versions are small relative to the total volume of data, the storage savings can be significant. In addition, alterations to the shared portions may also be shared, thus changing one data set can change the other correspondingly, if this is desired.

The recombine operation must be reconsidered in the light of sub-tree sharing. Recombination is not always desirable, even when an enfilade is badly balanced, since it would be an error to recombine a shared crum with an unshared one. The recombine operation must be carefully constructed, if used at all, in applications where sub-tree sharing is expected.

Another interesting (and pleasing) property of enfilades is that they

naturally and automatically tend to adapt themselves to take advantage of any clustering of the data in index space. This is because crums are grouped together into loaves according to the volume of material stored, rather than according to the material's indices. In addition, it is often the case that cuts will tend to occur in the less dense regions of the data structure. Over a period of time, the various manipulations performed will tend to bring about a helpful measure of correlation between patterns of sub-tree grouping in the enfilade and patterns of clumping in the data that it stores.

SUMMARY

An enfilade is a tree structure in which the nodes, called crums, do not directly store the indexing key, but rather a pair of localized abstractions of the key called the wid and the disp. These represent the extent and position relative to a parent in index space. Manipulation of an enfilade is supported by the operations retrieve, append, rearrange, insert, delete, recombine, cut, level push and level pop. Enfilades also allow sub-tree sharing.

(FOOTNOTE 1: In the Xanadu nomenclature, the term node refers to a computer system which is a member of a distributed data storage and communications network. The term crum is preferred for reference to a node in an enfiladic data structure. The extra term is introduced to avoid confusion, since the full-scale Xanadu design involves both enfilades and computer networks.)

Xanadu Hypertext System Data Structures

The current Xanadu backend is constructed using three essential data structures:

1) the granfilade, used to implement the grandmap

2) the poomfilade, used to implement orgls

3) the spanfilade, used to implement the spanmap

A fourth data structure, called the historical trace enfilade, used to realize the historical trace facility, has been designed but not yet implemented. In addition, alternate designs for the spanmap and the orgl, called the drexfilade and the DIV poom respectively, are anticipated. All of these data structures are described in this document. This document also describes another important aspect of the Xanadu implementation: the numbering system, tumblers, used to address entities in the system and the compressed number representation, humbers, that allows tumblers to be stored to arbitrary precision with reasonable efficiency.

TUMBLERS AND HUMBERS

A form of transfinitesimal number called a tumbler is used frequently throughout the system. Tumblers are like the numbers used to identify chapters, sections, sub-sections, pages, paragraphs and sub-paragraphs in many technical manuals. They are represented externally as a sequence of integer fields separated by periods (".") and internally as a string of integers. For example, 3.2 and 47.23.137.0.5 are tumblers (the period is delimiter for the human reader and is not essential to the fundamental notion of what tumblers are).

A non-commutative arithmetic operation called tumbler addition is defined. For example:

```
  3. 5.10. 6
+ 2.16. 3
  ---------
  5.16. 3
```

This arithmetic gives different roles to the two tumblers: the first specifies a position -- where something is -- and the second specifies an offset -- a distance to move forward. In this example, only the first field of the original position matters. A couple more examples are:

```
  25. 6.46.93              0. 0. 3. 1. 21
+  0. 0. 3. 1. 21       + 25. 6.46.93
  ---------------          ---------------
  25. 6.49. 1. 21          25. 6.46.93
```

The following rules describe the procedure for tumbler addition:

1) Evaluating from the most to the least significant fields (i.e., from left to right), the fields of the final result are equal to the corresponding fields of the initial position as long as the

corresponding fields of the offset are zero.
2) The first non-zero offset field is added to the corresponding field of the initial position.
3) The remaining fields of the final result are equal to the remaining fields of the offset.

Since tumbler addition is non-commutative, there are two possible forms of tumbler subtraction. We call these strong tumbler subtraction and weak tumbler subtraction. The Xanadu system makes use of a generalized tumbler difference operator defined as follows: when taking the difference of two tumblers A and B, if A is greater than B then the result is obtained by strongly subtracting B from A; otherwise the result is obtained by weakly subtracting A from B. As a result, no subtraction operation is ever performed that results in a negative tumbler. Like tumbler addition, tumbler subtraction involves two operands with different roles: a position and a negative offset. The difference between the two forms of tumbler subtraction is that strong subtraction results in a tumbler which may be added to the negative offset to get back to the original position, whereas weak subtraction involves simply applying the same essential procedure as tumbler addition with field subtraction substituted for field addition.

Here are some examples of strong tumbler subtraction:

```
   1.4.3              0.1.2.3.4.5          0.1.2.3.4.5.6
s- 1.1.1           s- 0.0.1.2.3.3       s- 0.1.2.3.3.3.3
   -----              -----------          -------------
   0.3.3              0.1.2.3.4.5          0.0.0.0.1.5.6
```

Note that:

```
   1.1.1              0.0.1.2.3.3          0.1.2.3.3.3.3
 + 0.3.3            + 0.1.2.3.4.5        + 0.0.0.0.1.5.6
   -----              -----------          -------------
   1.4.3              0.1.2.3.4.5          0.1.2.3.4.5.6
```

The following rules describe the procedure for strong tumbler subtraction:

1) Evaluating form the most to the least significant fields (i.e., from left to right), the fields of the final result are equal to zero as long as the corresponding fields of the initial position and the negative offset are equal to each other.
2) The first non-equal field of the negative offset is subtracted from the corresponding field of the initial position.
3) The remaining fields of the final result are equal to the remaining fields of the initial position.

Here are some examples of weak tumbler subtraction:

```
   1.4.3              0.3.3.3.4.5.6        0.1.2.3.4.5.6
w- 1.1.1           w- 0.1.1.3.3.3.3     w- 0.0.0.2.3.4.5
   -----              -------------        -------------
   0                  0.2                  0.1.2.1
```

The following rules describe the procedure for weak tumbler subtraction:

1) Evaluating from the most to the least significant fields (i.e., from left to right), the fields of the final result are equal to the corresponding fields of the initial position as long as the corresponding fields of the negative offset are zero.
2) The first non-zero negative offset field is subtracted from the corresponding field of the initial position.

Tumblers are used in the Xanadu system because of their accordion-like extensibility. They have the property that, like both real and rational numbers, between any two numbers are infinitely many more. Tumbler-space has a porosity that allows any amount of new material to be inserted at any point.

Tumblers are stored internally using Humbers. "Humber" is short for "Huffman encoded number" or "huge number". A humber is a form of infinite precision integer used to represent the fields of tumblers. Use of infinite precision integers prevents constraints on the size of a tumbler's field due to a restriction to, for example, 32-bit numbers.

Humbers are constructed out of 8-bit bytes. If the high-order bit of the first byte is 0, then the remaining 7 bits encode the number itself, with possible values ranging from 0 to 127. If the aforementioned bit is 1, then the remaining 7 bits encode the number of bytes in the number, and that many bytes then follow which contain it. Should more than 127 bytes be required to represent the number, the length zero (i.e., a high order bit of 1, followed by seven 0's) indicates that what follows is a humber (applying this definition recursively) that encodes the length, followed by the indicated number of bytes encoding the number.

This representation never runs out of precision and can represent any non-negative integer whatsoever (and, in fact can represent negatives with minor modification). In addition, most tumbler fields in the Xanadu system are small (i.e., less than 127) and thus can be encoded in a single byte. Tumblers are represented in a floating-point like format, with a mantissa consisting of a field count humber followed by the indicated number of field humbers, and an exponent humber indicating the number of leading 0 fields (leading since tumblers are transfinitesimals).

Tumblers are used in the Xanadu system as both V-stream addresses and I-stream addresses. We perform some tricks with these addresses by encoding some information about the thing addressed in parts of the tumbler itself. In particular we use fields with the value "0" as a delimiter to separate one part of a tumbler from another. For example, the "0" in 4.3.7.0.19.1 separates the tumbler into the two pieces 4.3.7 and 19.1. Of course, the pieces have to be constrained to not contain any "0"s themselves. Note also that the pieces are themselves tumblers.

Using the 0-field-as-delimiter scheme, an invariant orgl identifier is represented as a tumbler of the form:

<node>.0.<account>.0.<orgl>

where <node>, <account> and <orgl> are tumblers that don't contain "0" as one of their constituent fields. The <node> component is there in anticipation of future implementations which store orgls in a distributed network, and represents the particular node of that network with which the orgl is associated. <account> similarly anticipates future multi-user systems, and

represents the particular user or account (i.e., user associated with node number <node>) which owns the orgl. <orgl> indicates which particular one of that user's orgls is desired. For example, 5.0.3.2.0.17 indicates the 17'th orgl belonging to the 3.2'nd user on the 5'th node. In the present single-node single-user system, addresses all resemble 0.0.<orgl> with the <node> and <account> components being empty.

Both V-stream and I-stream addresses are tumblers of the form:

<invariant orgl id>.0.<v-space>.<position>

where <invariant orgl id> is tumbler representing an invariant orgl identifier of the form described above and <v-space> and <position> are integer fields. <v-space> specifies which virtual space of the orgl in question is desired, and <position> indicates a particular atom within that v-space. Thus 5.0.3.2.0.17.0.47.23 denotes the 23'rd atom of the 47'th v-space of the orgl specified in the previous example.

Use of the "0" field as a delimiter to concatenate the various components of addressing information into a single tumbler allows a single tumbler-space to contain all possible V-stream or I-stream addresses. This in turn simplifies the construction of data structures that use V-stream or I-stream space as their index space by eliminating any need for dealing with the various components separately.

Future plans may include support for an extended indirect V-stream address of the form:

<v-stream address>.<v-space>.<virtual position>

where <v-stream address> is a V-stream address of either this form or the previous one. This notation allows the immediate specification of atoms via orgls which are themselves within other orgls. For example, 5.0.3.2.0.17.0.47.23.2.1 would indicate the 1'st atom of the 2'nd v-space of the orgl addressed in the previous example (assuming, of course, that it was in fact an orgl and not a character atom).

The porosity of tumbler space enables new addresses to be created easily without conflict with previously created ones. For example, node 5 from the past few examples could create a new node 5.1 without requiring knowledge of whether, say, node 6 existed or not. Orgl ids for new versions of previously existing orgls are created this way. For example, the first new version of orgl 1.0.2.0.3 would be 1.0.2.0.3.1 and the next would be 1.0.2.0.3.2.

Physical storage locations are not, in general, represented using tumblers. Instead they have a form which is entirely dependent upon the nature and quantity of the underlying storage hardware and the dictates of the operating system software which interfaces with that hardware.

THE GRANFILADE

The granfilade is one of the three major data structures inside the Xanadu system, and probably the simplest. It is used to implement the grandmap, providing a means of looking up atoms by their I-stream addresses. As its name suggests, the granfilade is an enfilade.

The index space used by the granfilade is I-stream tumbler space. The wid of a granfilade crum is a tumbler specifying the span of I-space beneath the crum (i.e., the distance, in tumbler space, from the first to the last bottom crum descended from it). The widdative function is tumbler addition, therefore a crum's wid is simply the tumbler sum of its children's wids. Bottom crums have an implicit wid of 0.0.0.0.1 (i.e., spanning no nodes, no accounts, no orgls, no V-spaces and spanning a single atom). Granfilade disps are tumbler offsets in I-space from the parent crum.

Bottom crums in the granfilade represent atoms. Atoms come in two varieties: characters and orgls. Character bottom crums are stored as spans of characters, rather than individual bytes. These consist of a pointer to a block of physical storage, either in core or on disk, containing the bytes themselves, and an (integer) length, telling the number of bytes in the block. Since each of these bytes has an implicit wid of 0.0.0.0.1, the character span has a wid of 0.0.0.0.<number of characters in the span>. Orgl bottom crums are stored in a similar fashion, but instead of literal data bytes, pointers to the fulcrums of orgls (poomfilades) are stored.

To obtain the atom associated with a particular I-stream address, an enfilade retrieve operation is performed (see accompanying paper on enfilade theory). Starting with the fulcrum, the wid is subtracted from the desired I-stream address, and the child crum is selected whose disp comes closest to the modified I-stream address without going over it. The process is then repeated with this child crum and the modified I-stream address as the target. At the bottom, the (by now much reduced) target address is used as an index into the appropriate character or orgl span to select the desired atom. It is quite possible to find that there is no bottom crum associated with a given I-stream address, since tumbler space is porous (i.e., between any two tumblers are infinitely more tumblers).

Although the grandmap is said to simply map from I-stream addresses to physical storage locations, a retrieval operation on the grandmap may involve looking up the disk locations of the atoms in the granfilade, bringing the atoms themselves from those locations into core, and then returning the core addresses. This is something more than a simple lookup operation.

In practice, retrieves are performed upon a whole span at once, returning a number of atoms (which is specified in the retrieve request) starting at the given index space location. The length of the span to be retrieved is also a tumbler, due to the porosity of tumbler space.

The only operations performed upon the granfilade are retrieve and append, although there are an infinite number of potential places to append to (one such place for each possible V-space or each possible invariant orgl id). The granfilade is never cut, deleted from or rearranged. It simply accretes data monotonically, and then allows it to be quickly regurgitated. For this reason, it is quite reasonable to store the bottom and interior crums of the granfilade on an unchangeable medium, such as write-once optical disks.

THE POOMFILADE

The second major data structure inside the system is the poomfilade. The poomfilade is used to implement orgls. POOM stands for Permutations On

Ordering Matrix. The poomfilade represents something functionally similar, though not identical, to a permutation matrix: a sparse binary matrix in tumbler space, one axis of which represents the V-stream and the other the I-stream. A "1" at a given coordinate indicates that the orgl realized by the matrix maps that coordinate's V-stream axis location to and from its I-stream axis location. A "0" indicates that no such mapping exists. This would be a permutation matrix but for virtual copies, which result in a mapping between a single I-stream location and a number of V-stream locations, allowing a given I-stream "column" of this matrix to have more than one "1" entry. In addition, the previously mentioned porosity of tumbler space results in an infinite number of "rows" and "columns" which are all "0"s.

The index space used in the poomfilade is two-dimensional cartesian tumbler space. The disps and wids of poomfilade crums are ordered pairs of tumblers representing the positions and extents, respectively, of rectangles in this space. The positions (disps) draw their origin relative to the parent crum. The widdative function is construction of the minimum enclosing rectangle of the rectangles represented by all the children in a loaf.

Bottom crums in the poomfilade are identity matrices, representing V-spans that map without internal alteration onto I-spans. These are stored as a position (disp) along with a single value for extent (since identity matrices are, by definition, square). It is not necessary to actually store the "1"s and "0"s themselves.

Retrievals are performed on the poomfilade using one axis as the indexing axis and the other as the data axis. Extracting information consists of recursively delving into the boxes represented by the rectangles crossing the rows or columns of interest. A rearrange performed along the V-stream axis implements the corresponding primitive orgl edit operation (see the accompanying paper on the Xanadu system architecture) on the V-stream order of the orgl that the poomfilade represents. The other edit operations are implemented by manipulating the poomfilade in a similar fashion. Virtual copies are implemented using sub-tree sharing.

THE DIV POOM

The DIV poom is an extended design for the poomfilade that is planned but not yet implemented. The DIV poom is similar to the poom, but adds a third dimension to enable the determination of the orgl of origin of material that has been virtually copied. The third dimension is orgl (or "document", for historical reasons, hence the "D") of origin of the I-span. Bottom crums are still planar identity matrices, but associated with each is a position on the I-stream corresponding to orgl of origin for the span represented. Upper crums represent three-dimensional bounding rectangular prisms, although the enfilade is otherwise the same as the basic poomfilade.

THE SPANFILADE

The Xanadu system's third major data structure is the spanfilade. The spanfilade is used to implement the spanmap, a mapping from the I-stream to the O-stream (the subset of the I-stream whose atoms are orgls).

The spanfilade is similar to the poomfilade in its higher level structure

-- crums represent recursively nested boxes, and the widdative function is still minimum enclosing rectangle. The bottom crums are different, however, as is the interpretation of the axes.

The two axes represent the I-stream and the O-stream. In addition, the data structure is partitioned along the O-stream axis into independent segments, each of which spans the entire O-stream but represents the spans referenced from a particular V-space (this partioning is like having a set of parallel spanfilades, one for each V-space). Such a partitioning enables retrievals to be readily restricted to orgls from particular V-spaces.

Bottom crums represent I-spans referenced by a particular orgl in a particular V-space. A Bottom crum contains an O-stream position (relative to its parent crum), representing the I-stream address of a referencing orgl, an I-stream position (also relative to the parent crum), representing the start of an I-span, and an I-span length. The first two of these together constitute the bottom crum's disp, while the third is the crum's wid.

Retrievals are performed on the spanfilade using I-spans as the indexing elements. The result of such a retrieval is a set of I-stream addresses corresponding to the orgls which map some segment of the V-stream onto the I-span used as the key.

THE DREXFILADE

The drexfilade is a much improved design for the spanmap which will replace the current one as soon as possible. The drexfilade is also similar in flavor to the spanfilade, the poomfilade and the DIV poom (in fact, these are all part of a family which we call N-dimensional enfilades, where, in this case, N is 2 or 3). It is three dimensional binary matrix, with the three dimensions representing the O-stream and the starting and ending I-stream addresses of I-spans. Bottom crums are single points (the "1"s of the matrix). The interpretation of this matrix is as follows: a point is "1" (i.e., present in the data structure) if and only if the I-span it denotes on the two I-stream axes is present in (i.e., mapped to in full by) the orgl denoted by its O-stream axis position.

The drexfilade avoids some unfortunate combinatorial problems with the spanfilade. It also allows queries about arbitrary patterns of overlap between particular I-spans and the spans mapped to by the orgls, in addition to queries about simple enclosures.

THE HISTORICAL TRACE ENFILADE

The historical trace enfilade is certainly the most complex Xanadu data structure. Implementation has therefore been deferred until the more basic portions of the system are fully functional.

The underlying representation of an orgl is a sparse binary mapping matrix. Each of the primitive editing operations which the system supports -- rearrange, append, insert and delete -- is expressed as a transformation matrix that the poom is multiplied by to obtain the new, edited orgl. Note that the initial state of an orgl is a single identity matrix. When an edit operation is applied to this identity matrix, the resulting orgl is the transformation

matrix for that edit itself. From this it may be seen that each edit transformation matrix is in some sense an orgl itself, mapping from the old V-stream order of the orgl to the new V-stream order.

The matrix product of several of these transformation matrices is itself a transformation matrix for a single transformation that is equivalent to the constituent transformations applied individually. The product of all the transformations in the history of an orgl (not including alternate versions) is in fact the orgl itself.

The history of a phylum consists of the sequence of changes which have been made to its orgls over time. In the case of a phylum for which no versioning has taken place, this is a linear series of operations. Whenever a new version is introduced, the sequence of changes branches and becomes a tree. One way of visualizing this is as a wire frame tree with beads strung along the wires. Each bead represents a single primitive change to an orgl. It is this tree which forms the index space of the historical trace enfilade. There are thus two trees to keep in mind: the enfiladic tree -- the data structure itself -- and the tree-shaped index space in which the enfilade resides.

Locations in the index space -- points in the history of a the phylum -- are expressed as a path through the history tree. The tree is structured so that there are only binary branches. A path can be represented by distances (in units of single edit operations) interspersed with direction changes. If, with a binary tree, we follow the convention of always taking, say, the right branch unless otherwise directed, a path can be represented as a series of distances, between which are implicit left turns.

An enfilade is constructed in this space by grouping crums over regions of the tree (see the attached diagram). Each of these regions has a single entrance and zero or more exits. A crum's index wid is the set of paths through it to farther siblings. Its index disp is the path to its entrance relative to the entrance to its parent. The index widdative function is path concatenation.

The bottom crums of the historical trace enfilade are transformation matrices for the individual edit operations. These are quite simple. The index wid of a bottom crum is a single step along the historical trace.

In addition to its index wid, each crum also has a data wid. In the case of a bottom crum, this is the primitive transformation matrix that the it stores. In the case of an upper crum, the data wid is the matrix product of the data wids of its children. The data wid of a crum is thus the full transformation achieved by traversing the segment of the tree which it covers. This is in turn a poomfilade, constructed using sub-trees which are shared with the historical trace crum's children's data wids. The historical trace is thus an enfilade constructed from enfilades! Note that, because of branching in the historical trace tree, a crum may have a number of matrices in its data wid -- one for each path across the it. There is a one-to-one correspondence between the matrices in the data wid and the paths in the index wid. The data disp of an historical trace crum is implicitly the matrix product of the data wids of the crum's siblings along the path between the crum and the entrance to its parent.

A retrieval on the historical trace is accomplished by multiplying together the data disps of the crums descended through on the way to the bottom

crum located at a particular point on the history tree. The result is a poom for the orgl that existed at that point in the phylum's history. In practice it is not necessary to actually multiply whole matrices together, since the object of interest is not the final matrix itself but the mapping which it represents. It is sufficient to simply map whatever V-spans are desired through each of the disp matrices. This is equivalent to multiplying the matrices, but only requires dealing with the particular spans of interest, rather than the whole product orgl.

Xanadu Hypertext System Implementation

This document describeshe state of the current Xanadu implementation as well as a couple of other miscellaneous aspects of the Xanadu system design that have no other place to go: the frontend-backend interface and the core/disk memory model.

THE CURRENT IMPLEMENTATION (AS OF APRIL 1, 1984)

The system currently operates in a dedicated single-user mode. Support for multiple users/processes is anticipated as soon as funding permits. Multi-user Xanadu will be implemented as a transaction based system.

The present system supports an older interface paradigm in which orgls are separated into two classes called documents and links. Support for orgls in the sense that they are described in these documents will entail a change to the interface layer but not to the system proper.

The present implementation uses the older spanfilade and poomfilade designs (described above) rather than the newer drexfilade and DIV poom. In addition, as was mentioned earlier, historical trace has not yet been implemented.

Tumblers are currently stored internally using a fixed-size structure, rather than using the variable-length humber representation described here. This results in a substantial storage use inefficiency.

The single-user backend is written in C and runs under the Unix family of operating systems. The code, however, uses a minimum of operating system services and was written with portability in mind, so transfer to other systems is relatively straightforward. Multi-user will by necessity be somewhat more system dependent, but that limitation has not yet arrived.

THE FRONTEND/BACKEND INTERFACE

The Xanadu system makes a strong distinction between the data storage, retrieval and organizational functions and the user interface, display management and data formating functions. The latter are the responsibility of a separate frontend. The frontend is a program that runs as a separate process, possibly on a separate computer, and accesses the capabilities of the backend via some sort of data communications line or I/O channel. The backend implements the functions described in this set of documents in as application independent a fashion as possible.

The backend and the frontend interact via an interface language called Phoebe. This language allows programs to express requests to the backend to perform any of the functions described in these documents. The backend then responds with the requested data, or by performing the requested manipulation and returning an indication of whether or not it was able to do what was asked. The full interface language specification is not yet complete since our conception of the virtuality of the system has recently changed and the interface is being redesigned to take advantage of this. The interface protocol for the older virtuality is the one recognized by the current implementation. This protocol is described in one of the attached documents.

Phoebe will be described in detail in a future document.

THE CORE/DISK MEMORY MODEL

The present design assumes a two-level memory model: the hardware upon which the system is running has a limited quantity of high speed, random access memory which is immediately accessible, which we call core, and a much larger quantity of slower, less immediately accessible memory, called disk. The terms "core" and "disk" are chosen for convenience, and do not necessarily reflect the technology used to realize them.

Core is assumed to be a limited resource, restricted to a relatively small amount (e.g., a few megabytes per active process). Disk is not so constrained, and the model assumes that the amount of disk storage available is effectively unlimited. This is not to say that the system requires an infinite amount of disk space, but merely enough to contain all of the data that it is being asked to manage.

The contents of all of the system's data structures are stored permanently on disk. As used, they are also moved into core. All reading into core and writing out to disk is under the system's direct control. Swapping occurs at the data structure level, therefore use of a traditional demand-paged virtual memory system is not advantageous. The general rules for managing core storage are

1) all alterations to data structures must be performed in core,
2) if a crum is in core, then all of its ancestors must also be there,
3) within the above two constraints, try to keep as much material in core as possible.

Core is assumed to be less than totally reliable, in the sense that a system crash may cause any information stored there to be lost. Disk is assumed to be more reliable, and care is taken to ensure that the sequencing of transfers between core and disk preserves the integrity of data stored on disk. This in turn helps ensure that system failures leave the disk in a consistent state so that recovery is possible. The procedure for changing data stored on disk is: first, bring the data into core; second, perform the actual change; third, write a new copy out to disk; and finally, after verifying that what was written to disk was correct, write out to disk some indication that the new copy is now the active one.

The optimum procedure for managing core use in an environment that already has an underlying virtual memory mechanism is unclear. We feel that the Xanadu system itself can anticipate its own storage requirements and decide what to swap and when to swap it better than the algorithms used in typical demand-paged virtual memory systems. In some systems, however, it may not be practical to "turn off" the virtual memory. The best course in such a case is a matter for further study.

Xanadu Hypertext Virtuality

This document describes the Xanadu Hypertext virtuality -- the way it appears to outside users and the conventions we have established to make it appear that way.

The Xanadu virtuality is based on two primitive organizational structures: documents and links. A document represents some set of data, while a link represents a meaningful connection between some data in some documents.

A document consists of an ordered collection of characters together with an ordered collection of links. This is represented by an orgl. A document orgl's first V-space, called the text space, contains (only) characters. Its second V-space, called the link space, contains (only) links (described below). It does not have any other V-spaces.

A link consists of three ordered sets of spans of characters and links inside documents. These sets are called end-sets. The first of these three end-sets is called the from-set and the second is called the to-set. The link represents a directional connection from the material in the from-set to the material in the to-set. The third end-set is called the three-set and designates material pertaining to the nature of the link itself (i.e., what type of link it is or why it is there). Like a document, a link is represented by an orgl. A link orgl has three V-spaces, one for each end-set. These V-spaces are unconstrained as to what type of atoms they should contain. It is conventional, however, to structure link retrieval requests to ask the V-stream addresses of the contents of the end-sets, rather than asking for the atoms themselves. This is because a link represents a connection from one place (or set of places) to another, in addition to representing a connection from one set of atoms to another.

By convention, the first thing in a link's three-set represents a link type. Link types are represented by standardized, conventionally agreed upon V-stream addresses. The number of possible link types is unlimited, but a few standard types are defined by convention for the purposes of storing and organizing literary information. For example:

Jump link -- represents a simple connection from one place to another. A jump link indicates that such a connection exists. The major use of a jump link, as its name suggests, is to designate a possible jump from one body of material to another.

Quote link -- represents a quotation. The material in the to-set is embedded at the location defined by the first element in the from-set. This allows a quoted material to be expanded in its original context from quoted fragments.

Footnote link -- represents a footnote. The material in the to-set represents footnote-like commentary on the material in the from-set. The frontend may choose to display the to-set like a printed footnote by showing th text in the to-set at the bottom of the screen. The footnote link is intended to be used by the authors of documents to point t digressve matrial just as print footnotes are used.

Marginal note link -- represents a note "scribbled in the margin". A marginal note link is similar to a footnote link, but is intended to be used

by the readers of a document to point at reader created commentary on the material linked from. It may be desirable for frontends to disply marginal notes differently, as well. Also, it may be desirable to restrict retrieval of marginal note links to those with specific authors, whereas one probably wishes to retrieve footnote links in a less restricted manner.

Other useful link types are certainly possible, and what these should be is a matter to be settled between the designers of frontends and the users of system. Various kinds of semantic intent could be indicated by link types, in addition to regulating display and retrieval functions (e.g., by allowing the link type to indicate a perceived contradiction or agreement between the contents of the from-set and the to-set). Additional link types may also be desirable in order to support non-text or non-literary applications.

The model for this virtuality is an abstraction of what we feel are the fundamental underlying mechanisms of common paper-based literature systems (e.g., "the scientific literature"). Such generally consist of a corpus of writings -- documents -- in the form of papers, articles, book, tcialrrt, esonde explicit and implicit connections -- links -- such as citations, references, quotations, bibliographies, "in jokes", etc. "Electronic literature" is the primary application or which the Xanadu system was designed. It is our belief that, in order to supplant paper-based systems, electronic systems must preserve and enhance the capabilities already present in paper in addition to providing new capabilities hitherto not widely used.

Future Directions for Development of the Xanadu Hypertext System

This document describes future work to be done on the system to bring it from its current state as a fragile, partially functional single-user, single processor prototype to a robust, full-fledged, multi-user distributed system. Some of the tasks described here are necessary before the system will be minimally usable. Other tasks lead to more advanced functionality. The tasks are listed roughly in order of priority, but the actual order of implementation may vary depending on funding and customer priorities. Each task includes a description, a justification and a very rough estimate of the level of effort required to complete it, given the present type of development environment (programming in C under the Unix operating system).

Fix the various known bugs in the present system:

There are a few things that are just plain wrong and need to be corrected. In particular: links currently do not follow through to versions; there are some garbage collection glitches, especially when things get complicated; there used to be a persistent off-by-one error deep inside the code which was fixed and several places that worked by compensating for it themselves are now broken. At this time, none of these bugs are serious, in the sense that they interfere with the current capabilities of our demonstration system. They will become more important as work on the system proceeds. We estimate that these will be fixed in the course of debugging the system as a whole.

Modify to use variable-length tumblers:

The current fixed-size representation requires 40 bytes for each and every tumbler used in the system. Conversion to variable length will reduce this considerably. Most tumblers are quite small and in the variable-length format will require as few as 3 bytes. This will tremendously increase disk and memory efficiency. It will also speed things up because the amount of processing that may be performed in-core without swapping to disk will be increased. Also, any given operation on the data structures will have fewer bytes to shuffle. We have devised a plan for making the transition to variable-length tumblers in several independent steps. The first step involves modification of the low-level tumbler routines to handle tumblers in either format. Next, the various higher-level routines get converted one-by-one to use the variable-length format. Once the correctness of these modifications is verified, the fixed-length capabilities are finally removed from the low-level tumbler routines. This is a fairly involved process and may take a couple of man-months.

Install the drexfilade and the DIV poom:

These improved data structures are needed to permit realization of the full virtuality of our design. Fortunately, the higher level routines for N-dimensional enfilades, which the current spanfilade and poomfilade share, can also be used with little or no modification for the drexfilade and DIV poom. The lower level routines, which deal with bottom crums, and the routines which actually use the data structures to respond to requests will, of course, need to be modified. Conversion will be a two step process. The first step involves installing the new data structures and getting the system to work as it did with the old data structures. The second step involves adding the functionality which the new data structures provide that the old did not.

These two steps together will probably require several man-months.

Specify and implement the new frontend/backend interface protocol ("Phoebe"):

A new protocol is required to reflect recent important changes in the system virtuality. Before the beginning of this year the notions of document and link were deeply embedded in the system design. In particular, the current frontend/backend interface protocol reflects the restricted functionality provided by documents and links rather than the more generalized functionality provided by unconstrained use of orgls. The recent generalization to orgls somewhat reduces the range of possible requests that the frontend can give to the backend, but the remaining requests become much more complex. In addition, the conversion from the spanfilade to the drexfilade greatly increases the number of different kinds of restrictions that may be applied to retrieval operations.

Specification of a new protocol will involve an in-depth examination of the capabilities of the new virtuality in order to determine a set of requests that provides access to all of the system's capabilities without undue complication. Once the request set is determined, the format of the requests themselves and the grammar for the interface language will have to be designed.

Implementation of the new protocol will involve the construction of a new input parser for the backend and a nearly complete rewrite of those high-level routines directly responsible for fielding requests.

Both the specification and the implementation will be fairly large undertakings. We estimate a month for specification: a week to get the ideas sorted out, a week to figure out the request set, a week to design the grammar and a week to write it all down. Specification of the interface is a necessary precursor to much frontend work. Implementation will take somewhat longer: a few weeks to code and debug the parser and a month or more to rewrite the request handling routines. Implementation of the new protocol need not follow immediately upon completion of specification.

Optimize code around bottlenecks:

The system was implemented with a greater emphasis placed on things that could be readily made to work with the proper algorithmic efficiency, rather than on things that were efficient at a low level. As a consequence there are some pieces of code that are clear, correct and understandable and horribly slow. Some performance studies need to be done in order to find out where the real bottlenecks are. We have been hampered in conducting such studies by the state of the current release of Unix on our computers, in which the performance analysis and profiling tools do not work correctly. Performance studies have therefore been deferred until a newer release of the operating system software becomes available. The actual work of streamlining our system is an ongoing task. As with any complex system, there is a bottomless list of things that can be done to speed it up.

Implement the historical trace facility and design the protocol to use it:

Historical trace is a separable piece of the system, in the sense that the system will work without it. Implementation was therefore deferred until the rest of the system worked. Historical trace can be accomplished in two stages. In the first stage we build the historical trace tree with the

individual reversible edit changes at the bottom of the data structure but without the higher level aggregate matrices. This results in a simple branching edit log allowing retrieval of any version by tracing the individual edits in sequence from the current orgl. This works, albeit very slowly. In the second stage we erect the higher level data structure and speed things up, at the price of greater complication. The first stage, called slow historical trace, is relatively straightforward and can probably be accomplished in a month or two. The second stage, fast historical trace, is a major project and may require several months. Once slow historical trace is in place and the interface protocol is designed to use it, implementation of fast historical trace will be transparent to the user (except that it will be faster, of course). It is likely, however, to be somewhat challenging to implement.

Develop version archiving and "garbage collection" facilities:

Material stored in the Xanadu system accretes monotonically. Mechanisms must be developed to identify unreferenced or little used material and reuse the space it occupies while archiving it in some sort of low-cost long-term storage. Mechanisms to allow material to be permanently discarded may also be desirable. In addition, storage and time inefficiencies are introduced by orgl fragmentation as a result of substantial numbers of edit changes. One way of dealing with such fragmentation ist make a "cleancopy"of the material in the orgl at some time "t" and then provide an orgl mapping from V to V(t) (the V-stream order at time "t") in addition to the mapping from V to I. The major unresolved issue is the mechanism for deciding when a "clean copy" of some orgl should be made. Facilities such as these need to be both designed and implemented. For the first pass, we estimate a month to analyze the underlying problems and design solutions. The length of time required to realize the resulting design is unclear, but we guess that it would be on the order of a few months.

Implement multi-dimensional orgls:

In the present design, orgls represent linear (i.e., one-dimensional) collections of atoms. The generalization of the underlying data structures to higher dimensions is straightforward, but the implications of doing this are not clear. Further study is required to determine the desired virtuality for such a system and the protocols for controlling it. We estimate a few months of study and design work, and an unknown (but certainly greater) amount of implementation work.

Implement support for multiple-users:

This is essential. It is also the biggest single task in the list of tasks so far. The implementation of multi-user Xanadu will involve a greater degree of operating system dependence than is found in the present system. Our present notions of how to proceed are based upon 4.2 bsd Unix as the anticipated host operating system. Current plans call for multi-user Xanadu to be implemented as a transaction-based system. The idea is to split backend requests into series of the simplest possible primitive transactions which involve in-core data structure operations. These operations don't involve interaction with disk storage. Disk accesses are handled by Xanadu's virtual memory manager. A number of host dependent design issues are partially unresolved, and will remain so until the host environment is determined. These include:

-- What is the most appropriate memory management scheme for the

backend, given the host system's own underlying memory management? In particular, how do we go about sharing memory between processes?

-- What is the most appropriate way to deal with multi-processing in the backend? In particular, should the backend be split into several separate processes, and if so, how?

-- The complete mechanisms to handle berts must be designed.

-- Access protection and authority control mechanisms must be designed.

Once the complete multi-user system is designed, it must then of course be implemented. We estimate a month to study the host system, two months to design and specify the multi-user backend work, and six months to implement it. However, such time estimates are, at best, educated guesses at this stage.

Various design improvements and corrections:

There are a few things which we would like the system to do that the present design cannot. We have ideas as to how these various problems can be solved, but the ideas need to be refined. For example:

-- It cannot distinguish between the "original" of some material and a virtual copy if both the original and the copy are in the same orgl. This problem derives from the fact that virtual copies are made of I-spans, whereas they are referred to externally in terms of V-spans. The DIV Poom was the first step to solving this problem: in the original design, there was no way to distinguish between the original and the copy at all, even if they were in different orgls. The third axis of the DIV poom encodes the orgl of origin of a V-span (as opposed to the orgl of appearance). However, this does not enable us to determine where in that orgl the span came from, since this is potentially variable. Two possible ways of solving this problem are to change the way I-stream space is structured to embody a difference between copies of things, or to somehow dynamically keep track of how the V-stream address of a span shifts over time. Neither of these solutions is very well defined right now.

-- The system does not deal with the time dimension of material very well. In particular, we would like to be able to perform restricted retrievals using time as one of the dimensions of restriction. It seems like it would be relatively straightforward to time-stamp orgls as to their time of origin and most recent time of change. Adding time-wids to granfilade crums would also aid in filtering retrievals. As with the previous example, there appear to be things that we can do, but they need much more thought. Dealing effectively with the time dimension is something that we feel will be essential to many applications of the system.

-- There is undoubtedly a lot of tuning and refining that can be done to make things more efficient. We do not assume that our architecture is optimum.

These tasks are, by their very nature, rather vague and open-ended. Much of the work to be done in this category involves solving unsolved design problems. It is difficult to estimate how long this will take when the result is, by definition, not known. We do, however, have some ideas about how to proceed on some of these problems and feel that they are not insoluble. The design work listed here is perhaps best categorized as "ongoing".

Develop "semi-distributed" network support:

The "grand plan" for Xanadu involves a large, highly distributed network. This will not be readily accomplished, but we believe that an intermediate step to this, which we call "semi-distributed", is more feasible. Semi-distributed

networking involves distributing the functionality of a single-node Xanadu backend over several computers "tightly" coupled by a LAN. Each of the elements in this system would take responsibility for storing and managing parts of the different data structures, with one central machine acting as coordinator and request dispatcher. Orgls, being independent of one another, can be spread over several machines. The granfilade would reside on a central machine, though the data at which the granfilade points could be spread over several machines. The spanfilade, though it needs to be a centralized resource, could certainly be given a CPU of its own too. All of this is another large bite to chew. We guess a month or so to design it and many more to implement it.

Research into more advanced uses for enfilades:

The enfilade data structure paradigm used to realize the components of the Xanadu system has, we believe, additional uses. We have examined ideas for enfilades that would be helpful in such diverse applications as computer graphics, air traffic control, molecular engineering and relativistic physics. While the degree of rigorous analysis that has been performed in any of these fields is minimal, informal examination suggests that a closer look would be fruitful. At the present time this is an ongoing low-level effort and will undoubtedly continue as such.

Develop "fully-distributed" network support:

"Fully-distributed" networking implies large numbers of loosely coupled systems each functioning without centralized control and without necessarily having full knowledge of the complete network topology. This clearly involves solving a lot of problems that are at the forefront of research into distributed processing. Much further development work will be required before fully-distributed networking is possible. We don't care to even try to guess how long this will take, if it is possible at all. It <u>is</u> a problem we would like to work on.

We feel that much of the present work being done in the field of distributed data processing does not address the fundamental problems of network-based storage. Mainstream research in this field seems to concentrate on the problems of deadlock avoidance and the maintenance of data integrity. The Xanadu mechanisms of berts and versioning side-step these problems entirely. We feel that the hardest problems in distributed data storage have to do with the questions of how a node in the network determines where non-local data is to be found and how it determines how best to route its queries through the network in the absence of less than perfect knowledge of the network layout. In the Xanadu system this involves the problems of determining the existence and locations of remote links to, and remote versions of, local orgls and keeping track of who elsewhere knows about changes made locally. Our enfilade paradigm hints at some potential solutions to these problems: wids and disps may be viewed as a mechanism for representing partial knowledge. Some sort of enfiladic structure might be constructed to allow a Xanadu network node to maintain a representation of the overall network topology and of the contents of other nodes without requiring that the node have complete knowledge of everything that is happening elsewhere. We are fond of an analogy to the Hindu "Web of Indra", an infinite, intricate latticework of jewels in which each facet of each jewel holds a reflection of the whole lattice, including the reflections of the whole in the faces of all the other jewels.

Analysis of the Memory Usage and Computational Performance of the Xanadu System

ABSTRACT: Well golly. It's log. What else is there to say?

This document analyzes the theoretical performance of a multi-user Xanadu backend, constructed according to our present plans, in terms of memory (core and disk) consumption and computational requirements. It also discusses, briefly, the performance of the present implementation and how this performance differs from the most current design.

Memory/CPU Overhead

The current design for a multi-user Xanadu system calls for multiple processes in the backend processor to share a common segment of core memory which willcontain the in-core portions of the enfiladic data structures. Modifications to these data structures are permitted only under the umbrella of serializing transactions. Each transaction represents a single primitive operation on the data structures and is executed in an uninterruptable block of computation during which time the other processes are excluded from looking at or themselves modifying the system enfilades. Reading from disk is performed by Xanadu's underlying virtual memory manager when a process attempts to read a piece of some data structure which is not in core. Writing to disk is also the job of the virtual memory manager, and occurs when unused portions of the data structures are swapped out of core to make room for things being read in or when a process requests the virtual memory system to checkpoint its state.

The data structures required in-core for system operation are the granfilade (including the atoms its bottom crums date, the spanfilade and some number of poomfilades. However, only certain parts of these data structures need be core-resident at any given time. An operation performed on the data structure reads or modifies certain bottom crums and their ancestors, and only those branches of the trees which are are currently being used need be kept in core.

Enfilades are, in general, said to be logarithmic with the number of data items stored, in terms of both time to perform the various primitive operations and in terms of data structural storage overhead. Like many generalizations, this one is not completely correct. Storage overhead is logarithmic. However, certain multi-dimensional enfilades have non-logarithmic factors in the times required for certain useful retrieve and edit operations. These are discussed in more detail in the companion paper on enfilade theory. The worst case involves a term varying as the square root of a measure of orgl size, but we are not sure of the consequences of such non-logarithmicities in terms of system performance on typical documents. We believe that they are not a significant source of inefficiency.

Our memory management scheme requires that crums in core (i.e., the "active" crums) be accompanied by all their ancestors. The volume of the ancestor crums, as mentioned above, grows as the logarithm of the number of bottom crums in the whole system. Thus, core requirements vary as the product of the number of bottom crums in use with the logarithm of the total number of bottom crums in the corresponding enfilade(s). The amount of disk swapping over time will be directly proportional to the amount of change in the set of active bottom crums,which is effecvy constant for a given number of users.

The regularities of enfilades and their associated algorithms invite the

design of a memory management system that takes explicit account both of disk blocks and of the (somewhat different) tree-structures in core. Thus, we think that Xanadu's virtual memory manager can make more astute judgements about what to swap to disk than can a more general page-oriented virtual memory scheme. If Xanadu is installed on a system with such a virtual memory mechanism underneath, then it will need to know how much core it really has to work with. We are not sure how our virtual memory scheme would interact with those of list-oriented systems, since we are not current in our knowledge of list-oriented virtual memory techniques. We will undoubtedly learn about these if we work such systems to any significant extent.

Subtree sharing is used in multi-versioning orgls so that the common portions of different versions are stored in common. We believe that different versions of a given piece of material will have significant commonality and thus that subtree sharing will substantially reduce memory consumption. The exact degree of savings realized depends on the degree of commonality between versions which in turn depends upon the data being stored and on how the versions differ. Tradeoffs involving subtree sharing can better be studied when a fully operational system is available, since we can then see how people actually use it.

I/O Overhead

The Xanadu system makes use of two types of input/output (I/O). These are disk-block I/O, for data retrieval and long-term storage (via the virtual memory mechanism), and character I/O, for interaction with the frontend.

The frontend/backend interface is designed to minimize backend character I/O. The backend transmits requested data to the frontend in a terse format and does not concern itself with screen management or data display functions, since these typically require substantial computational overhead and I/O bandwidth.

The Current Implementation

Although the present system implementation has the basic algorithmic efficiency described above, it is inefficient in many low-level ways. We believe that, in general, optimization is best left until one has a working system to optimize. Many of the data structures now contain data which is redundant or stored in a wasteful manner. For example, it is common to use a full-word integer -- four bytes -- to store a one-bit flag. This would be changed in a production system, but makes sense in a development system because it simplifies the code by eliminating the need to twiddle individual bits. This in turn makes it easier to get the system working in the first place. Many data objects which are inherently variable in size are now stored in fixed-size chunks of memory which are typically much larger than required for the average object. For example, all tumblers are now 40 bytes long, but a typical tumbler will only occupy a fraction of this space in a variable-length implementation.

A single C struct is used to represent the upper crums of all three types of enfilades. This struct occupies 220 bytes in core and 164 bytes on disk. For storage on disk, the crums are packed into a struct which represents an entire loaf. The disk loaf struct is 1004 bytes long and contains 6 crums. The packing factor was chosen to enable a loaf to be stored in a single 1024-byte disk block. A crum is read into memory together with its sibling

crums, since disk blocks are read as wholes. To read in a bottom crum thus requires up to 6*220=1320 bytes times the depth of the enfilade. This in turn is the base-6 logarithm of the number of bottom crums in the whole enfilade.

Granfilade bottom crums contain either characters or orgls. A character bottom crum may contain up to 800 characters. An orgl bottom crum contains a single orgl pointer. This is in turn a struct which can hold both a core address and a disk address. The disk address is the permanent location of the fullcrum of an orgl on disk. The core address is the location in core of the core-resident fullcrum, if one exists (i.e., if the orgl is in core). The core location is null if the orgl has not been brought in from disk. Although up to 800 characters may be packed into a granfilade character bottom crum, no effort has been made to similarly pack orgl bottom crums. This is another source of inefficiency that can be easily corrected when appropriate.

Spanfilade and poomfilade bottom crums differ little from upper crums, but (like granfilade orgl bottom crums) are not tightly packed when on disk.

Miscellaneous Issues

The present prootype implementation spends a signican frction of its CPU time in a few low-level routines that perform tumbler comparison and tumbler arithmetic; these account for between 30% and 50% of the system's total CPU usage. Though simple, these operations are heavily used. Further, they require a procedure call and return each time they are invoked, and involve data types which are not supported by the underlying hardware. These characteristics make the tumbler operations good candidates for implementation in microcode.

Another support function that consumes a lot of computational overhead is storage allocation and free-space management (involving both core and disk space). The present scheme is modeled on Unix's storage allocation facilities, but was implemented from scratch since the Unix facilities try to gobble up all of the available core in the system and then blow up when they run out of space. The Unix storage allocation mechanism also requires an excessive amount of storage overhead when, as in our system, very large numbers of small pieces (i.e., a few bytes each) are being allocated and freed with great frequency. Our present storage manager is modelled after Unix's, so that it is not an improvement over the Unix storage manager in this respect. There are many possible techniques for dealing with storage allocation that are better adapted to dealing with small pieces. For larger pieces, a scheme using a doubly-indirect index through a table to ease compation of storage may be most appropriate. This would resemble the system used in Smalltalk-80 (insert reference).

Xanadu Hypertext System Frontend/Backend Interface

This document describes the current (April 1, 1984) Xanadu frontend/backend interface language. This document also is in the form of a BNF for the language with annotations describing what the various pieces are for.

Formats for information exchanged between the backend and the frontend:

```
<wdelim>  ::= '\n'
```

The newline character is used throughout the protocol as a general purpose delimiter.

Tumblers:

```
<tumbler>       ::= <texp> <tumblerdigit>* <wdelim>
<texp>          ::= <integer>
<tumblerdigit>  ::= <tdelim> <integer>
<tdelim>        ::= '.'
```

Tumblers are denoted by period separated strings of integers.

Addresses:

```
<doc id>      ::= <tumbler>
<doc-set>     ::= <ndocs> <doc id>*
<link id>     ::= <tumbler>
<doc vsa>     ::= <tumbler>
<span-set>    ::= <nspans> <span>*
<span>        ::= <tumbler> <tumbler>
<spec-set>    ::= <nspecs> <spec>*
<spec>        ::= { 's' <wdelim> <span> } | { 'v' <wdelim> <vspec> }
                  /* v for vspec, s for span */
<vspec-set>   ::= <nvspecs> <vspec>*
<vspec>       ::= <doc id> <vspan-set>
<vspan-set>   ::= <nspans> <vspan>*
<vspan>       ::= <span>
<ndocs>       ::= <integer> <wdelim>
<nspecs>      ::= <integer> <wdelim>
<nvspecs>     ::= <integer> <wdelim>
<nspans>      ::= <integer> <wdelim>
```

Addresses come in various flavors. A <doc id> is the V-stream address of a document (i.e., an invariant orgl identifier in the new interface model). A <link id> is the V-stream address of an atom which happens to be a link. A <doc vsa> is a V-stream address of an atom inside some document (i.e., an ordinary V-stream address). A indicates a range of addresses and is denoted by a starting address tumbler and a length tumbler. <doc-set>s, <span-set>s, <spec>s, <spec-set>s, <vspec>s, <vspec-set>s, and <vspan>s are various sorts of collections of all of these

Stuff:

```
<vstuffset> ::= <nthings> <vthing>*
<vthing>    ::= <text> | <link id>
<text-set>  ::= <ntexts> <text>*
<ntexts>    ::= <integer> <wdelim>
<text>      ::= <textflag> <nchars> <char>* <wdelim>
<textflag>  ::= 't'
<nchars>    ::= <integer> <wdelim>
<nthings>   ::= <integer> <wdelim>
```

"Stuff" is the generic term for the various sorts of things that can be found in a document: text and links.

Link stuff:

```
<from-set> ::= <spec-set>
<to-set>   ::= <spec-set>
<home-set> ::= <spec-set>
<link-set> ::= <nlinks> <link id>*
<nlinks>   ::= <integer> <wdelim>
```

Links are generally talked about in terms of their end-sets.

Calls to the backend:

```
CREATENEWDOCUMENT ::= <createdocrequest>
  returns <createdocrequest> <doc id>

<createdocrequest> ::= '11' <wdelim>
```

This creates an empty document. It returns the id of the new document.

```
CREATENEWVERSION ::= <createversionrequest> <doc id>
  returns <createversionrequest> <doc id>

<createversionrequest> ::= '13' <wordelim>
```

This creates a new document with the contents of document <doc id>. It returns the id of the new document. The new document's id will indicate its ancestry.

```
INSERT ::= <insertrequest> <doc id> <doc vsa> <text set>
  returns <insertrequest>

<insertrequest> ::= '0' <wdelim>
```

This inserts <text set> in document <doc id> at <doc vsa>. The v-stream addresses of any following characters in the document are increased by the length of the inserted text.

```
DELETEVSPAN ::= <deleterequest> <doc id> <span>
  returns <deleterequest>

  <deleterequest> ::= '12' <wdelim>
```

This removes the given span from the given document.

```
REARRANGE ::= <rearrangerequest> <doc id> <cut set>
  returns <rearrangerequest>

  <rearrangerequest> ::= '3' <wdelim>
  <cut set> ::= <ncuts> <doc vsa>*
  <ncuts> ::= <integer> <wdelim>              /* ncuts = 3 or 4 */
```

The <cut set> consists of three or four v-addresses within the specified document. Rearrange transposes two regions of text. With three cuts, the two regions are from cut 1 to cut 2, and from cut 2 to cut 3, assuming cut 1 < cut 2 < cut 3. With four cuts, the regions are from cut 1 to cut 2, and from cut 3 to cut 4, here assuming cut 1 < cut 2 and cut 3 < cut 4.

```
COPY ::= <copyrequest> <doc id> <doc vsa> <spec set>
  returns <copyrequest>

  <copyrequest> ::= '2' <wdelim>
```

The material determined by <spec set> is copied to the document determined by <doc id> at the address determined by <doc vsa>.

```
APPEND ::= <appendrequest> <text set> <doc id>
  returns <appendrequest>

  <appendrequest> ::= '19' <wdelim>
```

This appends <text set> onto the end of the text space of the document <doc id>.

```
RETRIEVEV ::= <retrieverequest> <spec set>
  returns <retrieverequest> <vstuffset>

  <retrieverequest> ::= '5' <wdelim>
```

This returns the material (text and links) determined by <spec set>.

```
RETRIEVEDOCVSPAN ::= <docvspanrequest> <doc id>
  returns <docvspanrequest> <vspan>

  <docvspanrequest> ::= '14' <wdelim>
```

This returns a span determining the origin and extent of the V-stream of document <doc id>.

RETRIEVEDOCVSPANSET ::= <docvspansetrequest> <doc id>
returns <docvspansetrequest> <vspanset>

<docvspansetrequest> ::= '1' <wdelim>
<vspanset> ::= <nspans> <vspan>*

This returns a span-set indicating both the number of characters of text and the number of links in document <doc id>.

MAKELINK ::= <makelinkrequest> <doc id> <doc vsa> <from set> <to set>
returns <makelinkrequest> <link id>

<makelinkrequest> ::= '4' <wdelim>

This creates a link in document <doc id> from <from set> to <to set>. It returns the id of the link made.

FINDLINKSFROMTO ::= <linksrequest> <home set> <from set> <to set>
returns <linksrequest> <link set>

<linksrequest> ::= '7' <wdelim>

Thisrturns a lt of al lins which are (1) in <home set>, (2) from all or any part of <from set>, and (3) to all or any part of <to set>.

FINDNUMOFLINKSFROMTO ::= <nlinksrequest> <home set> <from set> <to set>
returns <nlinksrequest> <nlinks>

<nlinksrequest> ::= '6' <wdelim>

This returns the number of links which are (1) in <home set>, (2) from all or any part of <from set>, and (3) to all or any part of <to set>.

FINDNEXTNLINKSFROMTO ::= <nextnlinksrequest> <from set> <to set> <home set> <link id> <nlinks>

returns <nextnlinksrequest> <linkset>

<nextnlinksrequest> ::= '8' <wdelim>

This returns a list of all links which are (1) in the list determined by <from set>, <to set>, and <home set> as in FINDLINKSFROMTO, (2) past the link given by <linkisa> on that list and, (3) no more than <n> items past that link on that list.

RETRIEVEENDSETS ::= <retrieveendsetsrequest> <spec set>
returns <retrieveendsetsrequest> <from spec set> <to spec set>

<retrieveendsetsrequest> ::= '26' <wdelim>

<from spec set> ::= <spec set>
<to spec set> ::= <spec set>

This returns a list of all link end-sets that are in <spec set>.

SHOWRELATIONOF2VERSIONS ::= <showrelationrequest> <spec set> <spec set>
returns <showrelationrequest> <correspondence list>

<showrelationrequest> ::= '10' <wdelim>
/* this is a wild guess at the vague form of the response */
<correspondence list> ::= <ncorresponences> <correspondence>*
<corresponence> ::= <item> <item>
<item> ::= <doc id> <vspan>
<ncorrespondences> ::= <integer> <wdelim>

This returns a list of ordered pairs of the spans of the two spec-sets that correspond.

FINDDOCSCONTAINING ::= <docscontainingrequest> <vspec set>
returns <docscontainingrequest> <doc set>

<docscontainingrequest> ::= '22' <wdelim>

This returns a list of all documents containing any portion of the material included by <vspec set>.

NAVIGATEONHT ::= <navigateonhtrequest> <totally undefined>
returns <navigateonhtrequest> <totally undefined>

This re-edits a document to any point in its editing history.

Xanadu Hypertext Source Code Annotations

This document describes various aspects of the C source code for the current implementation of the Xanadu Hypertext System backend. This document is intended as a guide and reference for people examining the source and readers may find parts of it to be fairly meaningless without a copy of the backend source as a companion document.

CODING CONVENTIONS

The Xanadu backend is written in C and we try to follow a fairly small set of coding conventions.

Identifiers -- we use long identifiers for the names of functions and global variables. The purpose of long identifiers, in our estimation, is not merely to avoid the use of cryptic abbreviations but to enable an identifier to contain a complete phrase or sentence fragment describing the thing identified. For example, a function to retrieve end-sets from the spanfilade would be named retrieveendsetsfromspanfilade in preference to, say, retspanfends. The purpose of this convention is to make the code more self-documenting. Long identifiers are used in preference to descriptive comments at the beginning of each function. There are two reasons for this: first, comments tend not to be updated in the frenzy of debugging and thus the comments and the actual code tend to drift away from each other; second, such comments are tied to the definitions of functions rather than to their use, thus making the purpose of a function call obscure if the function name is obscure.

Unfortunately, the long identifier convention was not followed from the very beginning of implementation, and some older pieces of code contain functions with shorter, less descriptive names. Also, note that we do not use any contextual cues to indicate the separations between words in an identifier, such as underscores (retrieve endsets from spanfilade) or capitalization (retrieveEndsetsFromSpanfilade). The original justification for this was that an identifier is a single unit, not a bunch of separate words. As a practical matter, if we had to do it over again we would probably follow the capitalization-of-interior-words convention. However, we feel that changing the convention in mid-implementation would result in confusion.

Another side-issue is that in standard C in the Unix environment, only the first eight characters of an identifier are significant (and only the first seven for external symbols). To cope with this we use a preprocessor which prepends a unique sequence of characters to any long identifiers which disambiguates them within seven characters. Currently, this preprocessor is used in all the C software we write. Soon we will be converting to 4.2BSD Unix which supports long identifiers directly.

Formating -- we use a control-structure formating convention which is popularly called the "one true brace style". The following examples are illustrative:

```
if (youdontcutthatout) {
        illtellyourmother();
}


if (itdoesntwork) {
        fix();
```

```
        } else if (itworksrealgood) {
                sell();
        } else {
                publish();
        }


        while (thecatsaway) {
                themicewillplay();
        }


        for (hesa(); jollygood; fellow()) {
                whichnobodycandeny();
        }


        switch (swarlock) {

          case ofdynamite:
                goboom();
                break;

          case ofcoke:
                takea();
                break;

          default:
                thebankloses();
                break;
        }
```

Functions are formated according to the following pattern:

```
  typeofreturneddata
functionname (firstargument, secondargument)
  typeoffirstargument   firstargument;
  typeofsecondargument  secondargument;
{
  typeoffirsttemporaryvariable  firsttemporaryvariable;
  typeofsecondtemporaryvariable secondtemporaryvariable;

        bodyoffunction;
}
```

Another formating convention we often follow is to keep very long lines "as is", rather than splitting them up. This is partially due to the fact that nobody has been able to agree upon a consistent convention for splitting lines and partially to enable us to unambiguously locate things using "grep". It does, however, result in some ugliness if the code is displayed or printed on a narrow (i.e., 80 columns or less) device.

Use of types -- We follow the practice of defining new data types for any structs that are used. In addition, types are defined for common uses of

normal data types such as ints. Most type names begin with "type", for example:

typedef struct foobarstruct typefoobarstruct;

An historical artifact found in some of the code is that many types are defined using #define rather than typedef. These definitions date back to the initial phases of implementation in BDS C, a CP/M based C dialect which lacks the typedef construct.

Comments -- As a rule, we are sparing in our use of comments. It has been our experience that comments and the implementation have a tendency to drift away from each other (as mentioned above). Comments are reserved for explaining things that are difficult to make obvious from the structure of the code itself (i.e., things which can't be made self-documenting). These are things such as efficiency hacks for optimization, kludges of any kind, obscure debugging fixes, and peculiar data structure use. Comments are also used to block out pieces of code for debugging purposes. If, when debugging, some function is found to require revisions which are either substantial or subtle, it is our practice to "comment out" the offending piece of code, make a copy, and then modify the copy. This enables us to recover the old version if we really mess up the new one. To aid in this, our preprocessor supports recursively nested comments. This enables us to comment out sections of code, which may themselves contain comments, with impunity.

COMMENTARY ON THE PRESENT SYSTEM SOURCE CODE

The present backend consists of slightly over 12,000 lines of C. Most (perhaps 75%) of this bulk is devoted to various support operations such as storage allocation and disk space management. This code is divided into approximately 500 C functions spread over approximately 50 files.

The two versions of the system --

We maintain two versions of the backend. One is called xumain and the other is called backend. Backend is the true backend program which interacts with a frontend. Xumain is a standalone version for debugging purposes which prompts the terminal directly for the input that it would ordinarily get from a frontend. The expected format of this input is the same as the what a frontend would produce and xumain is therefor considered "user hostile". We are gradually phasing xumain out as our frontend becomes more functional and sophisticated. Both versions share all of their code except for the very top level (main) and the interface protocol I/O routines.

High-level structure --

Both versions have the same structure. The main routine calls some initialization routines and then enters a loop. Inside this loop it repeatedly gets a request from the frontend (or the user terminal in the case of xumain) and processes it. The actual processing is done by a series of semantic routines, one for each request in the interface protocol. The selection of which semantic routine is to be used is determined by a table indexed by request number (the request number is, syntactically, the first thing in any request). Each of the semantic routines has the same basic structure. First, a "get" routine is called which reads and parses the parameters appropriate to the given request (there are two sets of "get" routines, one each for xumain

and backend). The "get" routine builds the appropriate data structures to represent the information contained in the request. A "do" routine is then called which performs the actual request. There is also one "do" routine per request. Each of these do routines calls the appropriate high-level enfilade manipulation routines to accomplish the requested action. The "do" routine returns a data structure containing the appropriate information in response to the request. Finally, a "put" routine is called which packages the information in the appropriate format and sends it back to the frontend (as with the "get" routines, there are two sets of "put" routines).

Storage management and virtual memory --

The largest portion of the code is devoted to storage allocation and deallocation. We found it necessary to write our own storage allocator because the one Unix provides could not cope with our application. In particular, there is no way to tell when you have run out of space when using the Unix storage routines. We also must manage the allocation of disk space. The present implementation simulates a raw disk by storing everything in a single giant Unix file named "enf.enf".

The coredisk virtual memory mechanism uses a cyclical garbage collector called the grimreaper. All of the data structures resident in core at any particular time are linked together in a large circular list. Associated with each piece is an age. When something must be swapped out to make room for something being swapped in, grimreaper traverses this list. Things which are sufficiently old are swapped out or discarded (depending upon whether or not they have been altered since they were brought in from disk). Things which are not old enough to be reaped have their ages incremented. Grimreaper continues traversing this list until enough space is freed to meet the present demands.

Another data structure used internally and which is not described elewhesa hi calaask. A task is simply a handle on a linked together collection of allocated temporary core storage associated with some function or task (hence the name) in the system. Temporary storage is allocated to a particular task. When finished with the activity to which the task structure belongs, the entire body of allocated temporary storage can be easily deallocated by traversing the allocation list in the task structure.

Creeping abbreviationism --

In spite of our long identifier convention, certain standard abbreviations have crept into some function names. The expansion of these is as follows:

```
pm  == poomfilade
sp  == spanfilade
gr  == granfilade
cbc == core bottom crum
cuc == core upper crum
dbc == disk bottom crum
duc == disk upper crum
nd  == n-dimensional enfilade (e.g., the poomfilade or spanfilade)
seq == sequential enfilade (e.g., the granfilade)
dsp == disp
vsa == v-stream address
isa == i-stream address
```

Other dangling cruft --

The present code contains a lot of ugliness due to debugging efforts. Often, pieces of code are commented out which are duplicate versions of troublesome routines. In addition, there are diagnostic print statements and calls to data structure dumping routines that clutter many places. These my somtimes obsure the true purpose of the code they are found in, especially if they are heavily used. There is also a whole file full of diagnostic routines which test various functions and allow complicated data structures to be dumped to the screen or to a file in a readable format. The routines are useful and important but add to the bulk of the code.

Glossary of Terms

This document is a comprehensive glossary of all the new terms introduced in the Xanadu documents. Terms are listed in alphabetical order, followed by their definitions. Little effort has been made to avoid circular definitions: for more complete explanations, read the relevant documents. Etymological notes are given in some cases for historical value and the amusement of the reader. These are enclosed in brackets ("[" and "]").

append -- One of the fundamental operations on enfilades. Adds new data items to an enfilade at the "end" of some dimension of index space.

atom -- One of the fundamental primitive entities that the Xanadu System deals with. There are two types of atoms: characters and orgls.

backend -- The component of a Xanadu System responsible for managing the actual storage and retrieval of atoms.

bert -- A locking identifier associated with the top of an orgl. Identifies the particular version of an orgl being dealt with when that orgl has been changed but not assigned a permanent address. Also prevents deadlock between processes by allowing concurrent access to documents. [Berts are named after Bertrand Russell, because they represent a fanatical effort to keep things consistent.]

bottom crum -- A crum at the bottom level of an enfilade. May be different regular crums since it may contain actual data that the upper crums do not contain.

core -- The term we prefer for a computer's local high-speed random access memory. [We like the term "core" even though it is archaic because 1) the term "RAM" is misleading, since disk is random access too; 2) the term "semiconductor memory" is as implementation technology specific and as likely to eventually become archaic, as well as being an unwieldy mouthful of words; 3) the term "core" is short, pithy and easy to remember; 4) everybody knows what you mean anyway.]

core/disk memory model -- The memory model used in the Xanadu System architecture which assumes a limited amount of high speed core memory coupled with large quantities of disk storage.

crum -- A "node" (in the traditional sense of the term) in an enfiladic data structure. Contains the wid and disp. [Named after a river in Pennsylvania on the banks of which the crum was invented. Also an acronym for "Chickens Running Under Mud" (don't ask).]

crum-block -- A loaf. Usually used in the context of packing large numbers of crums together in a disk block for long-term storage.

cut -- One of the fundamental operations on enfilades. Makes splits in the data structure for purposes of insertion, deletion or rearrangement. Cut is also a noun referring to a split made in the data structure by the cut operation.

data disp -- A disp whose purpose is not the location of items in index space

but rather the implicit containment of data itself.

data wid -- A wid whose purpose is not the location of items in index space but rather the implicit containment of data itself. A data wid is generally an abstraction or generalization of all the data stored beneath its crum.

data widdative function -- A function for computing a crum's data wid from the wids and disps of its children.

delete -- One of the basic operations on enfilades. Removes material from a specified portion of the data structure.

disk -- The term we prefer for a computer's lower-speed high volume storage. [The term "disk", like "core", is preferred because of its simplicity and brevity, though it may not be totally accurate.]

disp -- One of the two principal components of a crum (the other being the wid). Indicates the crum's displacement in index space relative to its parent crum. ["Disp" is short for "displacement" but has come to be its own term, rather than an abbreviation, since in some sorts of index spaces it may not be a displacement per se.]

DIV poom -- An extended form of the poomfilade that adds an additional dimension to represent orgl-of-origin. ["DIV" stands for "Document-Invariant-Variant", the three dimensions of the DIV poom. Document is a historical term for one conventional type of orgl.]

document -- One of the standard types of orgls in a literature-based Xanadu application. A document is an orgl with two V-spaces: a "text space" and a "link space".

drexfilade -- An improved model of the spanmap. [Named after Eric Drexler, the person who invented it.]

end-set -- Generic term for the collections of V-spans connected by a link. [The most simplified notion of a link is as a "magic piece of string" from one piece of data to another. End-sets are what are found at the ends of the string.]

enfilade -- One of a family of data structures characterized by being constant depth trees with wids and disps, rearrangability and the capacity for sub-tree sharing.

footnote link -- One of any number of possible standard link types. Represents a footnote.

four-cut rearrange -- One of two "flavors" of rearrange operation. Four cuts are made in the enfilade and the material between the first two is swapped with the material between the second two.

from-set -- The first end-set of a link. Contains the set of V-spans at the starting end of the directed connection represented by a link.

frontend -- The component of the Xanadu System responsible for user interface and any functions which depend upon the content of the data stored in

the backend (e.g., keyword searches).

frontend-backend interface -- The frontend and the backend communicate with each other in an interface language called Phoebe over some sort of communications line or I/O port. The frontend asks the backend to do things for it and the backend responds to these requests via the frontend-backend interface.

fulcrum -- The very topmost crum of an enfilade, from which all other crums are descended. [So called because it "contains" the full enfilade beneath it. Also, enfilades are frequently illustrated graphically as broad based isoceles triangles with the fulcrum at the peak (which does look like a fulcrum).]

grandmap -- One of the primary components of the system. Maps from I-stream addresses to the physical locations where the corresponding atoms are stored.

granfilade -- One of the primary data structures in the system. Used to implement the grandmap. ["Granfilade" implies "grand enfilade". It is the largest single data structure, in the sense that it "contains" everything stored in the system.]

historical trace -- A more advanced (as yet unimplemented) facility of the Xanadu System which enables the state of an orgl at any point in its edit history to be determined.

historical trace enfilade -- The enfilade with stores the history of a phylum so that the state of any of its orgls at any time may be reconstructed.

historical trace tree -- The branching pattern of changes and versions made to a phylum over time.

humber -- A form of infinite precision integer that can represent any number in a reasonably sized space. ["Humber" is derived from "Huffman encoded number".]

index disp -- A disp whose primary purpose is the location and identification of data items, as opposed to a data disp.

index space -- The space in which an enfilade "lives". Locations in this space are used as the index values identifying things to retrieve and the places to make cuts. An index space may have multiple dimensions, where a dimension is defined in our context as simply a separable component of indexing information which may be used by itself in a sensible fashion to (perhaps only partially) identify or locate something.

index wid -- A wid whose primary purpose is the location and identification of data items, as opposed to a data wid.

index widdative function -- A function that computes a crums index wid from the wids and disps of that crums children.

insert -- One of the basic operations on enfilades. Adds material at some specified location in the data structure. This operation is redundant

since it may be implemented by an append followed by a rearrange.

invariant orgl identifier -- A tumbler which is both a V-stream address and an I-stream address which identifies a "top" level orgl (i.e., one that is directly accessible from the external world rather than being retrieved as the contents of some other orgl.

invariant part -- The portion of a V-stream address which constitutes an invariant orgl identifier that identifies the orgl which maps the V-stream address to some I-stream address. It is a syntactically separable part of a V-stream address.

invariant stream -- The address space in which atoms are stored. When initially placed in the system, each atom is assigned to the next available space on the invariant stream.

I-span -- A span of atoms on the I-stream. Consists of a starting position together with a length. The term "I-span" is variously used to refer to the addresses in such a span or to the atoms themselves, depending upon context.

I-stream -- Abbreviation for "Invariant stream". Used more commonly than the longer term.

I-stream address -- A location on the I-stream.

I-stream order -- The order in which atoms appear on the I-stream.

I-to-V mapping -- The correspondence between I-stream addresses and V-stream addresses which is represented by an orgl.

jump link -- One of any number of possible standard link types. Represents the simplest possible connection from one place to another.

level pop -- One of the fundamental operations on enfilades. Makes the data structure smaller (in both actual and potential size) by removing a redundant fulcrum.

level push -- One of the fundamental operations on enfilades. Enlarges the potential size of the data structure by adding a level on top of the fulcrum.

link -- One of the standard types of orgls in a literature-based Xanadu application. A link is an orgl with three V-spaces called "end-sets": the "from-set", the "to-set" and the "three-set".

link space -- The V-space of a document orgl which contains links.

loaf -- A group of crums together. Generally used in the context of the group of sibling crums that are the set children of some other crum. [A loaf is of course what you get when you pack a bunch of crum(b)s together.]

marginal note link -- One of any number of possible standard link types. Represents the connection to a "marginal note".

N-dimensional enfilade -- A family of enfilades whose index spaces are

N-dimensional euclidean spaces indexed by cartesian coordinates.

node -- A computer in a distributed processing and data storage network.

orgl -- One of the primary data structures in the system. Maps from V-stream addresses to I-stream addresses and vice-versa. ["Orgl" is short for "ORGanizationaL thingie".]

Phoebe -- The name of the frontend-backend interface language. [Phoebe is derived from "fe-be" which in turn is short for "frontend-backend".]

phylum -- The collective group of orgls represented by an historical trace tree. [The term "phylum" denotes a tree-like family structure. It also sounds vaguely like "file".]

POOM -- A permutation-matrix-like mapping which is implemented by the poomfilade. Used to represent orgls. ["POOM" stands for "Permutations On Ordering Matrix".]

poomfilade -- The enfilade used to represent POOMs and therefore orgls.

process -- A particular connection to the backend that may request the storage or retrieval of characters and orgls.

quote link -- One of any number of possible standard link types. Represents a quotation.

rearrangability -- One of the properties of enfilades. Rearrangability means that pieces can be reorganized on one level of the tree and descendant levels will automatically be reorganized accordingly.

rearrange -- One of the fundamental operations on enfilades. Changes the order in index space of material. There are two types of rearrange operation called "three-cut rearrange" and "four-cut rearrange".

recombine -- One of the fundamental operations on enfilades. "Heals" cuts and compacts the data structure after it has been fragmented by other operations.

retrieve -- One of the fundamental operation on enfilades. Obtains the data item associated with a particular location in index space.

span -- A contiguous collection of things, usually characters. Usually represented as a starting address together with a length. The term is also sometimes used to refer to the length itself (as in "what is the span of this document?").

spanfilade -- One of the primary data structures in the system. Used to implement the spanmap. ["Spanfilade" implies "enfilade for dealing with spans".]

spanmap -- One of the primary components of the system. Maps from the I-stream addresses of atoms in general to the I-stream addresses of orgls referencing those atoms.

sub-tree -- Some portion of an enfilade denoted by a crum and all of its

descendants. A sub-tree is itself an enfilade with the crum its peak as the fulcrum.

sub-tree sharability -- One of the properties of enfilades. Sub-tree sharability means that sub-trees can be shared between enfilades or between different parts of the same enfilade in a manner that is transparent to the fundamental operations that can be performed.

sub-tree sharing -- The act of taking advantage of sub-tree sharability.

text space -- The V-space of a document orgl which contains character atoms.

three-cut rearrange -- One of two "flavors" of rearrange operation. Three cuts are made in the enfilade. The material between the first and second cuts is swapped with the material between the second and third cuts. This can also be seen as moving the material found between the first and second cuts to the location defined by the third cut, etc.

three-set -- The third end-set of a link. Contains material the addresses or contents of which indicate something about the nature or type of the link. [The term is a pun, counting the end-sets "from, to, three" (one, two, three).]

to-set -- The second end-set of a link. Contains the set of V-spans at the terminating end of the directed connection represented by a link.

tumbler -- The type of number used to address things inside the Xanadu System. A tumbler is a transfinitesimal number represented as a string of integers, for example "1.0.3.2.0.96.2.0.1.137". ["Tumbler" is sort of a contraction for "transfinitesimal humber".]

tumbler addition -- A form of non-commutative addition defined on tumblers for purposes of, among other things, implementing the ".+." operator for enfilades constructed in tumbler space.

upper crum -- A non-bottom crum.

variant part -- The portion of a V-stream address which identifies a particular location inside the orgl identified by the invariant part. It is a syntactically separable part of a V-stream address.

variant stream -- The address space in which, to the outside world, atoms appear to be stored. Also called the "virtual stream" or "V-stream". [See "V-stream".]

version -- An alternate form for some orgl, representing a past, future or current-but-different organization for the same body of material.

versioning -- The process of storing alternate organizations for a given body of material by constructing multiple enfilades which use sub-tree sharing for the portions where they are the same.

virtual copy -- 1. A copy of some set of atoms made not by duplicating the atoms but by mapping additional V-stream locations onto the atoms' I-stream locations. 2.Duplication of some portion of a data structure by using sub-tree sharing rather than by actually copying the material

stored.

virtuality -- The "seeming" of something. "Virtuality" is to "reality" as "virtual" is to "real". [Virtuality is one of innumerable terms coined by Ted Nelson. Since it is a useful concept, the term has stuck with us.]

virtual space -- A separately addressable sub-region of an orgl. An orgl may have any number of virtual spaces.

virtual stream -- The address space in which, to the outside world, atoms appear to be stored. Also called the "variant stream" or "V-stream". [See "V-stream".]

virtual stream address -- A location on the virtual stream.

V-space -- Short for "virtual space".

V-span -- A span on the V-stream.

V-stream -- Short for "virtual stream" or "variant stream". [The "V" variously stands for "virtual" or "variant" because of the trade secret status of much of the Xanadu internals: "Virtual" is the public term, referring tc the order in which things appear to the outside world. "Variant" is the private term, referring to the relationship between the Variant (i.e., changing) order that is shown to the world and the Invariant (i.e., not changing) order in which things are actually stored.]

V-stream address -- Short for "virtual stream address".

V-stream order -- The order in which atoms appear on the V-stream.

V-to-I mapping -- The correspondence between V-stream addresses and I-stream addresses which is represented by an orgl.

wid -- One of the two principal components of a crum (the other being the disp). Indicates the crum's width in index space (i.e., the volume of index space spanned by its children). ["Wid" is short for "width" but has come to be its own term, rather than an abbreviation, since in some sorts of index spaces it may not be a width per se.]

widdative function -- The function, characteristic of any particular type of enfilade, which computes a crum's wid from the wids and disps of its children. A notable property of the widdative function is that it is associative.

Xanadu -- The name of our favorite hypertext system. [Taken from the poem by Samuel Taylor Coleridge about a mythical paradise constructed by Kubla Khan.]

.==. -- Notation for operator that tests for the "equality" of two index space locations.

.<. -- Notation for operator that tests whether one index space location "precedes" another.

.<=. -- Notation for operator that tests whether one index space location is "less than or equal to" another.

.+. -- Notation for index space "addition" operator.

.-. -- Notation for index space "subtraction" operator.

.O. -- Notation for the origin of the index space.

Description of tasks and considerations for Xanadu development plan

(Task names begin with capital letters (i.e., "Document VM"). Considerations begin with lower case letters (i.e., "origin of link spans"). Tasks whose descriptions are not self evident from their titles are explained in greater detail following the task name.

A. Design

Includes all those tasks which involve figuring out the broad outlines of how the system is to be put together and what the external form of the system should be. Does not include detailed implementation design (that is considered a part of the implementation task).

1. Host system training

Familiarize ourselves with the idiosyncrasies of the tools we are going to be using throughout the project.

a. Understand Interlisp

Learning our way around the Interlisp environment: how to compose, compile and debug programs and how to use the various tools that are lying around.

b. Figure out their VM

Obtain a solid working understanding of the underlying virtual memory mechanism in the Interlisp-D system.

2. Virtual Memory

Involves designing the backend virtual memory system to support Xanadu's needs while coexisting harmoniously with the (already present) virtual memory system of the host environment.

a. Integrate with host VM

Figure out how to construct our VM on top of Interlisp's, without clashing with it and hopefully being assisted by it.

b. Design LRU scheme, working sets, list VM

Design the basic mechanisms of our virtual memory.

c. Design coredisk, free disk allocation

Work out some of the crufty details of our VM.

d. Document VM

3. New data structures

The various considerations associated with this task (items a-d) are pending design problems to be solved in the new data structures.

a. origin of link spans
b. virtual copy fix
c. link span info in granf
d. exfoliating trees

e. Document new data structures

4. Time stuff

Figure out how to cope with the time dimension of material stored in the system. Items a-d are considerations.

a. requirements for retrieval
b. time-stamp stuff
c. time sieving
d. delta-t wids in HT, elsewhere
e. times of interest: create, modify, read (first, last, what parts when?)
d. retrieve by? primitives; time restrictions

g. Design time stuff

5. Future plans and considerations

Involves those aspects of the future Xanadu design which must be built into the single-user system to avoid having to redesign or reimplement the system from scratch when we undertake development of a multi-user distributed system. This task essentially consists of making sure that we are not stepping on our own toes.

a. Design modularization for multi-user

Determine the modular structure of a multi-user system so that the modular structure of the single-user system will not differ from it significantly.

b. Figure out modularity for semi-distributed (single-node, multi-CPU)

Design the architecture of the future semi-distributed system in order to find any ramifications that such plans might have on the current single-user design.

c. Figure out communications and synchronization for semi-distributed

d. Consider ramifications of greater distribution (multi-node)

e. Design archiving; internode protocol

Design facilities for archiving versions of documents in order to cope with storage fragmentation when the volume of data stored in the system becomes very large. This includes some thinking about distributed systems since things that originate in one part of a distributed system might be archived in another.

f. Document future plans and considerations

6. Historical Trace

Design the virtuality of the historical trace facility.

a. HT black box spec

Determine the desired functional behavior of the historical trace facility.

b. HT protocol

Design the components of the frontend/backend interface protocol which deal with the historical trace facility.

7. Protection and authority control

Select and design the data protection, security and access control features of our system.

a. Design protection

b. Design authority control

8. Retrieve protocol
Design the frontend/backend interface protocol components for the retrieval of information from the backend.

a. Format and design of restriction sets

b. Bert internal and external representation

c. Design retrieve protocol

9. Backend design completion
Various miscellaneous things which must be accomplished before the backend design can be said to be truly complete.

a. Verify that disk output is safe from crashes
Make sure that the design is such that data integrity and the structure of orgls is preserved if the system crashes.

b. Document internal structures design

c. Document new protocols
Document the various pieces of the frontend/backend interface protocol that have been designed.

10. Test-Frontend
Design a frontend for purposes of testing and diagnosing the backend.

a. Design test-FE functionality

b. Design test-FE virtuality

B. Implement
Includes designing the details of the actual code to be produced and then actually producing it, including both coding and debugging.

1. Additional host system training

a. Really understand Interlisp
Learn about various more subtle aspects of the Interlisp-D environment that may effect the implementation effort.

2. Variable length tumblers
Produce the primitive routines to handle Xanadu's peculiar underlying data types.

a. Design tumbler routines

b. Implement tumbler routines

3. Virtual Memory
Implement the backend virtual memory.

a. Figure out how to implement shared core
This is necessary for future multi-user operation.

b. Design backend VM code

c. Implement backend VM

4. New data structures
Implement the backend data structures and the routines to manipulate them.

a. Unify enfilade routines with object model
Structure the data structures and the code which manipulates them so that one set of consistent enfilade routines may be used with a variety of different underlying enfilades.

b. Figure out ND orgls
Determine the generalization of orgls to multiple dimensions so that the present one-dimensional implementation will be a step on the path to a future multi-dimensional one.

c. Figure out implementation of in-core subtree sharing

d. Design code to handle data structures

e. Implement code to handle data structures

f. Document data structures' implementation

5. New frontend/backend protocol
Consists of implementing the parser and interpreter for the frontend/backend interface protocol. This is the component of the system which "drives" everything else.

a. Design parser for fe/be interface protocol

b. Design I/O routines

c. Design semantic routines for fe/be interface protocol

d. Implement parser

e. Implement I/O routines

f. Implement semantic routines

6. Backend semantic stuff
Implement the "high level" routines which perform all the actual data manipulations.

a. Design be semantic routines on top of enfilades

b. Implement be semantic routines on top of enfilades

7. Test-frontend

Implement the frontend for testing and diagnosis of the backend.

a. Screen management

b. Frontend VM

c. Command interface

d. Backend interface

e. Interface with world
Provide an interface between the functions of the backend and the functions of the Interlisp environment.

f. Link following/link handling

g. Retrieval functions

8. Convert single-user to multi-user
Most of the underlying aspects of the system required for multi-user operation will have been built into the single-user system. This task involves making the actual step to multi-user.

a. Connect multi-processes

9. Historical Trace
Implement the historical trace facility.

a. Implement transaction log

b. Design HT code

c. Implement HT

10. Security and accessibility
Involves various miscellaneous implementation tasks which must be completed before the system is truly ready to use.

a. Implement protection and authority control

b. Interface with higher-level network protocols
Implement code to assure that the system may be used as a network-based resource.

c. Verify that disk output is safe from system failures

11. Archiving
Implement long-term document and version archiving facilities of various sorts. Items b-d are considerations.

a. Implement version consolidation

b. tape
c. optical disk
d. internode

12. Document implementation
Produce the final documentation package for the single-node system.

a. Document pragmatic stuff
Document those aspects of the system which have crept in due to the underlying environment and due to other practical influences.

b. Final documentation

C. Test and measure
Analysis efforts required for the planning of further development.

1. Performance measurements
Perform various performance tests and benchmarks and analyze the actual performance of the implemented system. Items c-f are considerations.

a. Perform tests

b. Document performance measurement results

c. disk
d. cpu
e. response time
f. memory

D. Optimize
Make the system faster, more efficient, more compact, etc.

1. Optimize

a. Optimize

E. Advanced implementation
Additional implementation work to take advantage of the potential for multi-CPU distribution of the system in the special case of a LAN coupled environment.

1. Multi-user to Singe-node semi-distributed
Single-node semi-distributed involves distribution of various parts of a single-node over several machines.

a. Implement multi-user single-node semi-distributed system

2. Single-node Semi-distributed to Multi-node semi-distributed
Multi-node semi-distributed involves taking advantage of the rapid and reliable inter-node communications between a small number of computers which characterizes a local area network (LAN).

a. Figure out modularity for multi-node semi-distributed

b. Figure out synchronization and communications for multi-node semi-distributed

c. Implement multi-node (LAN coupled) semi-distributed system