Application Interface to the **LINCKS** Database System through the **LIBLINCKS** library.

Programmers Manual

Manual version: 1.1 1/28/94

Lab for Intelligent Information Systems
Department of Computer and Information Science
University of Linköping, Sweden

Author: Lin Padgham

Contributors: Ralph Rönnquist, Martin Sjölin, Leif Running, Patrick Lambrix, and David Partain.

Last updated: January 28, 1994

Abstract

This manual is a description of the library at the lowest access level of the **LINCKS** system. It should only be used as a temporary measure prior to the availability of a more 'friendly' application programmer's manual.

E-mail: lincks@ida.liu.se
Bug Reports: lincks-bugs@ida.liu.se

Contents

List of Figures

1 Introduction

This document describes the interface to the **LINCKS** database system through the set of functions available in the **LIBLINCKS** library. In order to use these functions, the application programmer needs a clear understanding of what is offered by the **LINCKS** data base, including a distinct definition of the necessary terminology.

The first sections of this document present a summary of **LINCKS** terms from the viewpoint of an application programmer. Section 2 is a general overview recollecting the basic concepts. Section 3 describes the data structures used in the communication between **LINCKS** and an application program. Section 4 describes the version handling, and section 8 contains an overview of the **LIBLINCKS** functions, followed by detailed descriptions of each function in Section 9. Section 5 describes the functionality for handling composite objects. Sections 10 and 11 list error codes and some caveats.

2 Some Concepts

An *atomic object* in **LINCKS** (see figure 1) is represented by two kinds of nodes: *versions* and a *history structure* node. The versions represent the appearance of the object at particular points in time. The history structure node contains the historical information of the object (see also section 4).

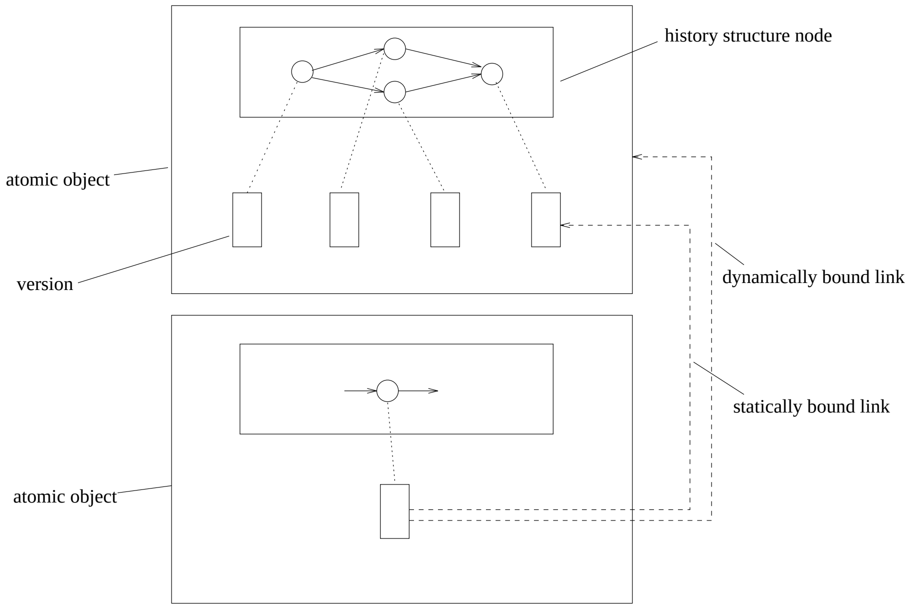


Figure 1: Atomic object in LINCKS

A node contains three sections: *attributes*, *links*, and *image*. The attribute and link sections are further divided into *groups*, *fields*, and *values*, while the image is only a value. Figure 2 outlines the hierarchy:

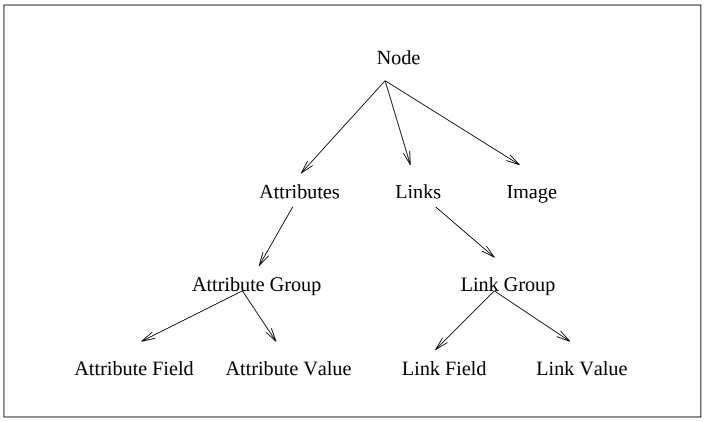


Figure 2: The Parts of a Node

Thus, for example, "attribute value" means the value of some field in some group in the attribute section of some node.

All but the three main sections (*Attributes*, *Links*, and *Image*) are "repeating fields" indexed by tags. We talk about "group tags" and "field tags". We have adopted a policy to use capital letters for group tags, while field tags are capitalised words.

The structure of link values is defined to be varying length arrays with start index one (1). Each entry in such an array is called a *link item*, and it refers to either an object or an object version. In the first case we call the link *unbound*. In the second case we have a *bound* link. (see also section 3.3)

An application program refers to objects through a data structure called a *label* (see also section 3.3). The labels should, generally, be treated as handles to objects and nodes, and thus be set and reset only through calls to **LIBLINCKS** functions.

3 Data Types

The following data structures are defined for communication between the **LINCKS** database system and an application program. As a general policy, both structure building and memory management is left to the application program.

3.1 Attribute Values

Attribute values are normally treated by **LINCKS** as blocks of memory. No attributes except the few listed below are dedicated for any particular use, and therefore the interpretation of attribute values is left to the application program. The following declaration (found in liblincks.h and sunlincks.h) should be used for communicating attribute values.

```
typedef struct {
    int attsize;
    char *attvalue;
} attrval;
```

The field *attsize* contains the block size in number of bytes, while the field *attvalue* contains the (lowest) memory address for the block.

The function **GA_GETATTR()**, which retrieves an attribute value, allocates a block for (a copy of) the value, and fills in the length and address of the block in the *attrval* record provided by the caller. The application program is responsible for *free*'ing the block.

3.2 System Defined Attributes and Links

The **LINCKS** system defines two attributes: **SYSTEM:Created**, and **SYSTEM:Owner**. The values of these are set for each version as soon as it is created. The value of **SYSTEM:Created** is the date and time of creation, and the value of **SYSTEM:Owner** is the login name of the user responsible for creating the version.

There is also one system defined link: **SYSTEM:Parent**. This link refers to the node version from which the present version was derived (i.e. the edit history (section 4)). The **SYSTEM:Parent** link often coincides with the temporal history (section 4), but is sometimes different. This reflects that some other than the most recent version has been edited, such as when going back to an older version and making a new start.

3.3 Labels

All **LIBLINCKS** functions have at least one label pointer as argument. The label structure is declared in liblincks.h and sunlincks.h as follows:

```
typedef struct {
    int vs;                 /* Version structure molecule index     */
                            /* Identifier for a set of versions
                             * of a node. */
#define VS_SYS_ROOT 0       /* System root node. */

    int inst;               /* Instance molecule index */
                            /* Identifier for a particular version
                               of a node, or indicator for unbound
                               reference. */
#define INST_UNBOUND (-1) /* Unbound link.  The old term was
                               "free reference". */
} label;
```

Labels should always be treated as handles to objects and nodes and should not be changed by the application program. The label fields can be read out as "*inst* version of object with *vs* as history structure node".

A label is either *bound* to some particular version of an object, or is an *unbound* reference to the object, in which case it dynamically refers to the current version. A bound label becomes unbound through a call to the function **UBL_UNBINDLABEL()**, while an unbound label becomes bound either as a side effect when it is passed in to **LIBLINCKS** functions that change node contents, or through a call to the **LIBLINCKS** function **GC_GETCURRENT()**.

The application program is free to assign labels to labels, and to compare labels. The labels should then be treated as pairs of integers according to above. Label comparison has intricate semantics though, due to the use of currency.

If two labels are equal in both *vs* and *inst*, they refer to the same version of the same object. This holds independent of whether they are unbound or bound.

If the labels differ in the *inst* field but not in the *vs* field and both are bound, they refer to different versions of the same object.

If the labels differ in the *inst* field, but not in the *vs* field and one of them is unbound, then they may still refer to the same version of same object. The unbound label may refer to it through the current link, while the bound label refers to it directly. In this case, it is advisable to bind the unbound label through a call to **GC_GETCURRENT()**.

If the labels are different in the *vs* field, they refer to different objects.

3.4 Lists

There are some **LIBLINCKS** functions that, logically, result in lists of various kinds. For instance, there is a function **GLV_GETLINKVAL()** that returns the value of a field in a group in the links section, which thus is a list of links. To handle such cases, we have adopted the policy to use functions that take a function pointer as argument. So, for example, the function

GLV_GETLINKVAL() is called with an argument *mapfn*, which is a function that gets applied once for each argument in the list. The application program can then form its own structure.

The exact form of the *mapfn* is documented where it is used. See section 8.7 for some example *mapfn*'s.

3.5 Image

The image section is, like attributes, treated as a block of memory, and the same access method is used.

3.6 Group Tags and Field Tags

All tags are null-terminated strings. Tag comparison is case dependent. **LINCKS** has internal tag buffers wherein tags are put when retrieved. The application program should always copy it to a desired place.

4 Version Handling and Currency

4.1 Temporal History

An object has a history structure node which allows access to previous versions of that object. Currently, the history structure node contains the *temporal history* of the object. The temporal history is a partial order over the versions representing the relation ‘is created before’. When two versions of an object are not related to each other with respect to this relation, we call them *parallel* versions.

Versions have a status. When a version is created in a workspace, but not yet stored in the database, we say that the version is *transient*. When the version has been stored in the database, the version is promoted to be a *working version*. A working version cannot be changed, i.e. when we try to modify a working version, a new transient version is automatically created.

The history structure node contains the following links for handling the temporal history.

- VSNET: the VSNET group contains fields which represent the temporal history involving the working versions. A field in this group has the form *datetag lbl1 ... lbln* where n$\geq$1. The datetag is a unique identifier. The labels represent the temporal history. In *datetag lbl1 lbl2 lbl3* for example the field asserts that both 'lbl1' and 'lbl2' are before 'lbl3' in the temporal history.

- TH_TRANSIENT: this group represents the temporal history involving transient versions. The fields in the group are similar to the fields in VSNET. When a transient version is promoted to a working version (by storing in the database), the field in TH_TRANSIENT involving this version is moved to VSNET.
It may happen that transient version are not stored in the database before logout. Then the content of this version are lost but the entry in TH_TRANSIENT involving this version still exists. These entries can be removed using **RTL_REMOVETRANSIENTLINKS()**.

- SYSTEM:Latest: represents the latest versions with respect to the temporal history. Observe that SYSTEM:Latest can be different in a workspace and in the database. SYSTEM:Latest in the database only considers working versions, while SYSTEM:Latest in a workspace also considers the transient versions created in that workspace.

- SYSTEM:Current: holds the version used for the dynamic binding mechanism.
This link does not control anything. SYSTEM:Current gets set upon each LL_bindlbl call or upon explicit **SC_SETCURRENT()** command.

- SYSTEMCURRENT:user: represents for user ‘user’ the version of that object that was edited latest.

- BUFFERNODE: holds the transient version for an object in the workspace. It has no meaning in the database.
- SYSTEM:Former_Latest: holds the versions which were latest since the retrieval of the object, but which are no longer members of SYSTEM:Latest. This field has meaning only in the workspaces, not in the database.

When a transient version is created (i.e. upon creation of a new object, or upon modification of an existing object having only working versions) the following happens:

1. TH_TRANSIENT and BUFFERNODE get a new field involving the new version.
2. A unique datetag is created.
3. BUFFERNODE gets a field with the datetag as fieldname and the version as linkitem.
4. TH_TRANSIENT gets a field with the datetag as fieldname and the version as last linkitem.
5. The elements in SYSTEM:Latest are put in the new TH_TRANSIENT field as linkitems, before the new version.
6. The elements in SYSTEM:Latest are added to SYSTEM:Former_Latest.
7. SYSTEM:Latest in the database doesn't change. SYSTEM:Latest in the workspace becomes SYSTEM:Latest in the database minus the elements of SYSTEM:Former_Latest plus the new version.
8. SYSTEMCURRENT:user in the workspace is set to be the new version.

When a transient version is promoted to a working version (i.e. by an explicit store command) then the following happens.

1. The field in TH_TRANSIENT involving this version is moved to VSNET.
2. BUFFERNODE is set to be empty.
3. SYSTEM:Latest (in both the database and the workspace) is set to be SYSTEM:Latest in the database minus the elements of SYSTEM: Former_Latest in the workspace plus the new working version.
4. SYSTEM:Former_Latest is set to be empty.
5. SYSTEMCURRENT:user in the database is set to be the new working version.

4.2 Edit History

The **LINCKS** system also maintains an *edit history*. This history represents the relation 'is used to create'. The relation is a partial order and can relate versions of different objects.

Currently the edit history is maintained by using the SYSTEM:Parent links in the versions. The linkitem in SYSTEM:Parent points to the version which was used to create this version. This link is automatically updated by a CPO_COPYOBJECT command or upon the creation of a new transient version.

5 Composite Objects

5.1 Concepts

5.1.1 Binding Tables

A binding table version has links to each of the components of a composition at a particular point in time. However, the bound links to components are not represented in the binding table version. (This means that to build up the complete composition, we need both the binding table and the presentation description object.) The history of the binding table represents the history of the composition, with each binding table version representing a version of the composition.

The binding table versions (see figure 3) have two special link names: 'COMPOSITION:Root' contains a version of the root of the composition. This might look something like `COMPOSITION:Root <313:24>`. 'COMPOSITION:Components' contains for each unbound component one version of the component. Again, this would have a series of labels such as `<214:12>,<215:4>`.

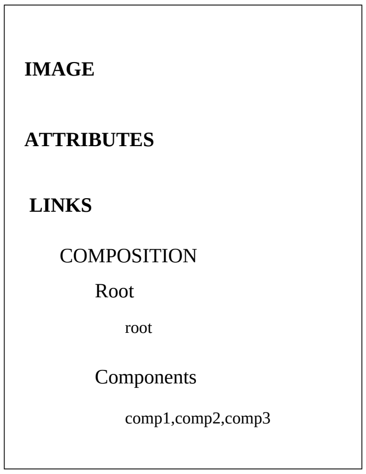


Figure 3: Binding Table Version

5.1.2 Root of composition

The root of a composition must contain the information which points to the compositions for which it is the root. Keep in mind that the root of a composition can be the root of several compositions.[1] In each root version there has to be a field 'COMPOSITION:Compositions', where the members in the linkitem list in

[1]The root of the composition is the object pointed to by 'COMPOSITION:Root' in a binding table version. However, the information in the binding table version is **not** used to determine the root of the composition, as that information is known to the AIM layer. The resolution only takes place in the other direction, from root of the composition to binding table.

the field in the odd places are presentation description objects, while the members in the even places are binding tables. (So we have actually a list of (presentation description object,binding table)-pairs.) (see figure 4). Thus, given the root of an object, we can then determine the binding table to use.

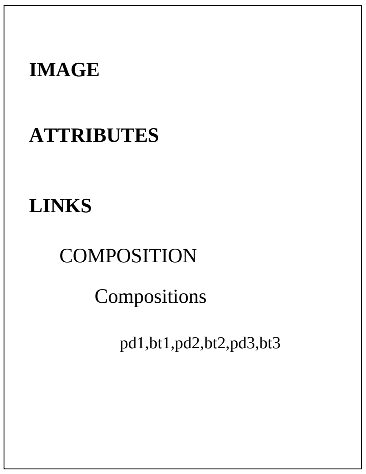


Figure 4: Root Version of a Composition

5.2 How to use the functions for composite objects

5.2.1 Access a composition

To access a composition one has to know the presentation description object version and the root version of the composition. To find the appropriate binding table, call *GBTV_Get_Binding_Table_Version*. If the composition existed before (i.e. the combination of the presentation description object and the root) then we get a binding table version as described above. To get the versions of the root and the component objects, call for each of them *GCRV_Get_Component_or_Root_Version*. Otherwise, i.e. the composition did not exist before, a new binding table with an empty transient version is created. In the latter case the (presentation description object, binding table) pair is added to the root but the root and components in the binding table must still be filled in. (One way to do this is to use the *ARBT_Add_Root_to_Binding_Table* and *ACBT_Add_Component_to_Binding_Table* functions. One could also use *GCRV_Get_Component_or_Root_Version*. The difference is that in the first way the component/root must be bound while in the second way the function allows unbound components/root. The function binds the component to be the 'best' version.)

5.2.2 Access a particular composition version

To find a particular composition version, on should access the composition as described above. Then one should search through the binding table versions to find the appropriate composition version.

5.2.3 Create a new composition

Creating a new composition is the same as accessing the composition for the first time (sse above). So call *GBTV_Get_Binding_Table_Version* with a presentation description object version and a root version of the composition to create a new binding table. Then add then the root and the components to the binding table.

5.2.4 Update the root

To update the root, first edit the root object. Then call the function *GCRV_Get_Component_or_Root_Version* to update the binding table.

5.2.5 Update a component

To update a component, first edit the component object. Then call the function *GCRV_Get_Component_or_Root_Version* to update the binding table.

5.2.6 Add a component

Adding a component, means linking or the new component to a component one level up (which can be the root). In this case the component one level up is updated and *GCRV_Get_Component_or_Root_Version* should be called with this component to update the binding table. Also *GCRV_Get_Component_or_Root_Version* should be called with the new component.

5.2.7 Delete a component

Deleting a component, means unlinking the component from a component one level up (which can be the root). In this case the component one level up is updated and *GCRV_Get_Component_or_Root_Version* should be called with this component to update the binding table.

5.2.8 Store a composition

To store a composition the root and all components should be stored. Although not strictly necessary, for efficency reasons later on, one can call the function *RRV_Remove_Redundant_Versions*.

6 Parallel Notification

Martin ????

7 Command History

7.1 Concepts

7.1.1 What is a command

In order to discuss command histories, we need first to be clear about what a command is, i.e. what it is that is kept track of with the command history. We therefore take the position that a command is an instantiated function that has been executed successfully. The information to store about a command is the function name and how it was instantiated, i.e. its parameters.

LIBLINCKS always operates on local copies of objects and changes are stored in the common database only with an explicit store operation. We therefore identify two different kinds of commands. The first are called workspace commands and the second database commands. Workspace commands operate on the local copies of objects while database commands operate on objects in the actual database; essentially store and get operations.

7.1.2 Command histories in general

We also need to answer ‘what do we mean with a command history?’, and more specifically ‘how do we want to use the command history?’.

The most basic question to answer with command history is ‘was command A executed before/after or in parallel with command B’. For this, we must have a temporal ordering relation between the commands. The ordering is partial, since different users can execute commands in parallel (and actually update the command history in parallel).

In figures, we present the command history as a graph where nodes are commands and arcs indicate temporal succession, see figure 5.

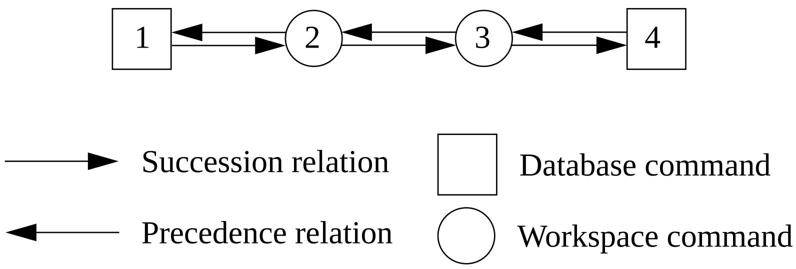


Figure 5: An example of a command history graph.

7.1.3 The relations

It is worth pointing out that the temporal order relationship between commands may be understood as a succession relation in the forward direction and as a precedence relation in the backward direction. Although there is no important difference between these views at the modelling level, it makes a significant difference which view is chosen for implementation through object linking. A forward direction linking more easily supports queries regarding what happened after a given command whereas the backward direction linking supports queries on what happened before a given command.

7.1.4 The session history

Commands are naturally grouped together into sessions so that each command represents one point in time during the session it occurs in. If a session is an object (in the sense of the LINCKS database model), then each command occurring during the session can be seen as an object version. In this way the database command history has a natural partitioning into session objects whose versions are the commands, and the building of a command history can utilise the existing temporal version management. See figure 10, chapter 4. ????

7.1.5 Multi user command history

Commands of different sessions, typically simultaneous sessions of different users, are also related temporally. This provides an additional dimension for temporal queries, and gives answer to questions like: 'Did user a execute command X before/after or in parallel with users b's execution of command Y' or 'Who changed document X, who changed document Y and which was done first'.

The links between commands of different sessions are always links between database commands. The workspace commands are, as a consequence of the workspace-database separation, invisible for other workspaces. Therefore it makes little sense to allow temporal links between workspace commands of different sessions.

So we will have two time structures in the command history. The first is the user session history that is stored as versions in a session history object. The second is a partially ordered time relation between the database commands, i.e. the commands that applied to the database, see figure 6.

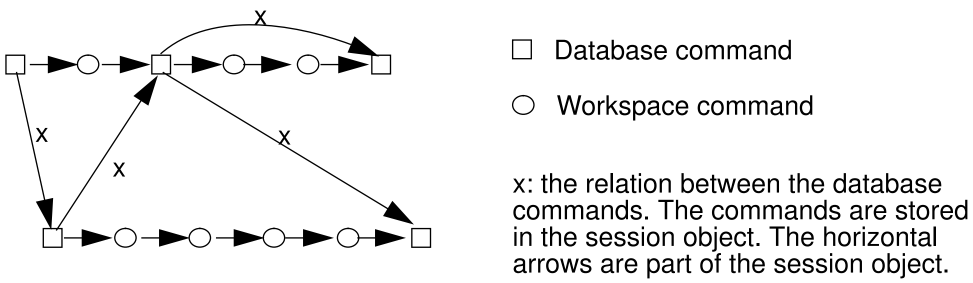


Figure 6: The two time structures.

7.1.6 Command history in LINCKS

In this implementation, the commands that are stored in the command history, are the editing commands of the **LIBLINCKS** interface. Also, two more abstract commands are logged: the composite store and expand; for operations on composite objects. See section 7.3 for a detailed list of commands logged in the command history.

The composite commands will work as a sort of delimiter. For instance, the composite store groups together the detailed store commands in the sequence that stores all newly changed parts of a composite object.

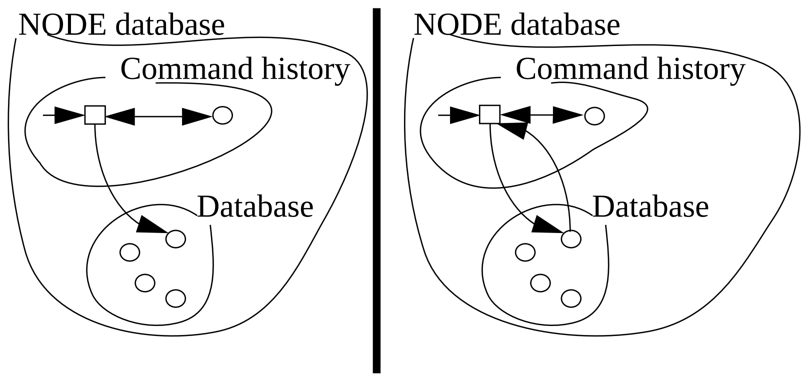


Figure 7: The integration command history - database

The commands in the command history are linked to the affected objects, and especially the object store command has a link to the object version stored, see figure 7. For this command, it is also useful to have the inverse link, i.e. a link from the object version to the store command, see figure 7. This allows for an easy way to compare the creation times of atomic objects, as well as of composite objects. We only store this backlink from the created version to the command history when the command is a store command.

The command history is itself contained in ordinary objects in the database, and thus requires some extra precautions in order to protect it from unauthorized editing.

7.2 Representation & Implementation

The command history is contained in a two level representation, where the commands issued within a user session are kept as distinct versions of a session object. Database commands are further ordered temporally by linking together the concerned versions. The temporal ordering between database commands will typically relate commands in different user sessions, thus providing a connection between different user session.

The fine-grained command history as contained within the session objects has a direct management support in the existing temporal history of the database. Session level connection management involves three specific problems:

1. to avoid unnecessary linking,
2. to allow parallel command history update as well as containing parallel commands in the history, and
3. to choose linking direction relative to temporal order, i.e. forward or backward in time, or maybe both.

A direct linking of two commands is considered unnecessary if they already are connected indirectly through a chain of links. The chain of links already provides, by means of transitivity, the information of temporal order, and therefore a direct linking is redundant.

Parallel update of the command history occurs whenever two, or more users store database commands in the history at the same time, since they both have to update the command history. Parallel update of the command history will result in parallel database commands in the history. If there are two commands in the history such that there are no chain of links from one of them to the other, then they are considered as parallel commands.

Linking direction is an issue for reasons of complexity. At first I investigated a forward linking approach, which required considerable computation for managing parallel updates. Later, I turned to a backward linking approach, and could reduce code size by an estimate of 30%. The backward linking approach is also conceptually simpler.

7.2.1 The session object

The session object is a sequence of command nodes that represents a user session from login to logout. The command nodes hold information about how the commands were instantiated and links to the affected objects. See appendix B for a complete list of command node attributes and links.

By storing a session as versions of the same object, we use the existing temporal history to represent the succession relation and the edit history to represent the precedence relation. See figure 8 for an example.

It is important to note that this way of using the versioning mechanism of LINCKS does not utilise the contents inheritance along the edit history. Normally, as an object version is created sub-sequent to an existing version, the new version first inherits the contents of the old version, and thereafter the new version may be edited (until explicitly stored into the database). This inheritance is not useful for the command history, but rather an obstacle, since every new session object version must be emptied after creation and before the appropriate command information can be filled in.

For instance assume that a command A is recently stored in the command history with the particular information that the attribute 'Group' has the value 'Chapters'. Assume further that the sub-sequent command B does not concern the attribute 'Group' at all. As the node to represent B is created as successor to A, it will, by virtue of the inheritance, automatically contain that attribute

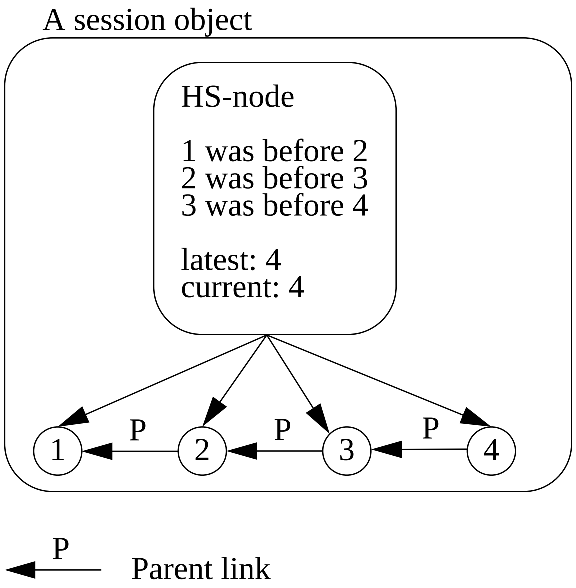


Figure 8: How the example history in figure

'Group' has value 'Chapters'. This information is however not relevant for B, and it must therefore be removed in forming the representation of the command.

We have used the session object despite the fact that it cause problems with the inheritance, because it gives a natural grouping of the sessions and we get the temporal relation as part of the object history.

7.2.2 Removing of redundant arcs

Between session objects, we set up links to handle queries like 'did user a know that user b had executed command Y when he executed X?'. These links are always between database commands, and are never connected to any workspace command.

One way to incorporate those links in the command history is to add arcs between the different single user command histories according to the following rule. Let X1,X2 to Xn be the different logged in users. Each of them then has one latest executed database command stored in his own session history, l1, l2 to ln. When user Xi executes a new database command, and stores that command in the command history, we add one arc from each of the commands l1- ln to the new command node li. See figure 11. We let a session be illustrated by a horizontal sequence of commands; circles for workspace commands and boxes for database commands.

Clearly this provides all information needed, but if carried out simple mindedly, it has the disadvantage that it may result in redundant arcs. For instance, assume that users A, B and C all have executed some commands, and then B executes a database command followed by C executing a database command. This gives a situation as showed in figure 10.

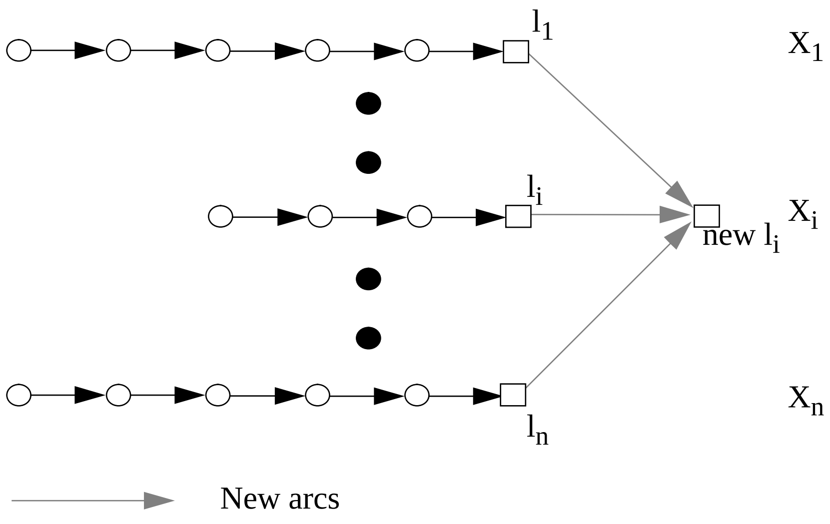


Figure 9: First attempt to a multi user command history.

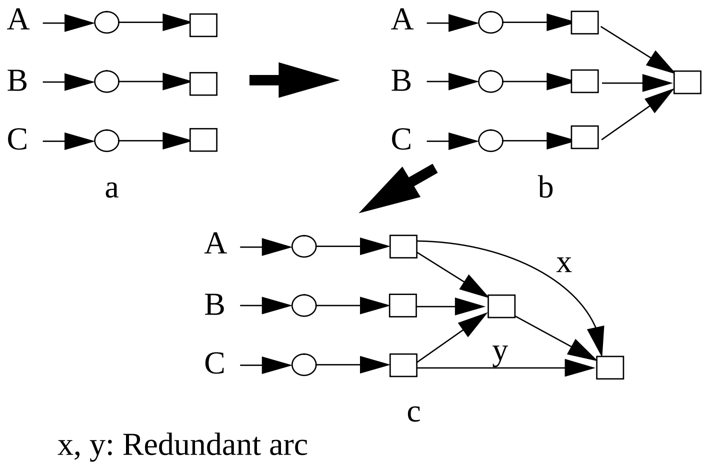


Figure 10: The problem of redundant arcs.

The arc x in figure 10 (c) is unnecessary to store, because it is a transitive arc and the same command node can be reached by following two arcs. The arc y is however useful to store since it is a part of user C's session. If we removed the y arc, we would break up the session, which would be unfortunate.

The present implementation handles arc redundancy as follows. We maintain a set 'Front' that contains all the latest database commands that do not have any other database commands executed after them. When a new database command executes, it is connected to the commands in 'Front', which thereafter is reset to contain the one command that was just executed, see figure 11.

The only control structure needed is the set of front commands, i.e. the database commands that have no database commands after them. To that end we use a control object, called ch object (Command History object), which is also the handle to the command history as a whole. Each version of the ch object has one

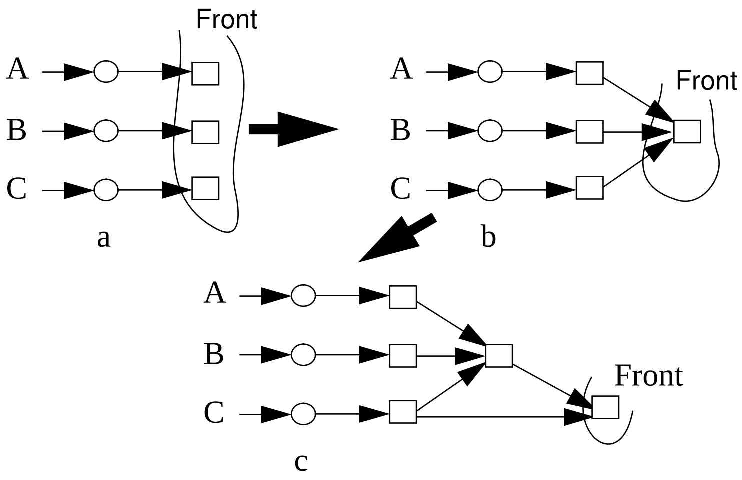


Figure 11: The solution to the problem with redundant arcs.

link 'Front' to exactly one front command node. The set of latest ch object versions, through their 'Front' links, forms the set of front commands in the command history, see figure 12.

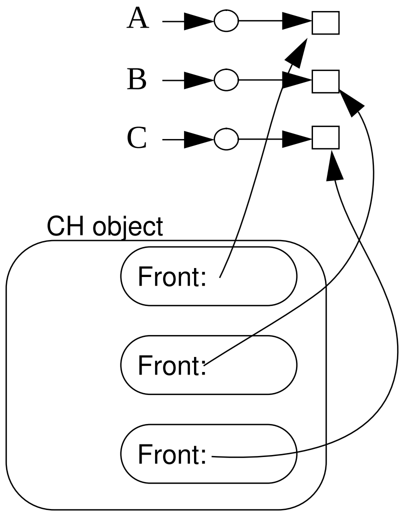


Figure 12: The Ch object for the command history in

7.2.3 Parallel updating of history

We need to ensure that we manage the situation when two, or more, users try to update the command history at the same time. Assume for instance, that after the situation showed in figure 13b, both user A and C execute some database commands in parallel. Then we want to get the command history shown in figure 13.

This is however not a problem, since it is solved by using the LINCKS feature that the editing is done on a local copy and then stored back to the database with

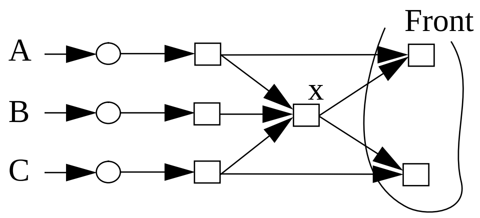


Figure 13: Parallel commands.

an explicit store command. Both users get the same initial ch object; actually the same set of latest versions of the ch object. They each add their new database command in their session object and add links from the commands in the set 'Front' to the new command. The 'Front' set will in this case be only the command x. The set of latest version of the ch object will, by each user, then be reset to contain only one version, the one with a front link to the new command. The two new versions of the ch object are then stored in the database, which result in two versions in the set of latest versions.

7.2.4 Linking

When it comes to linking the database commands together there are two approaches, forward and backward linking. A forward direction linking more easily supports queries regarding what happened after a given command whereas the backward direction linking supports queries on what happened before a given command. My initial attempt used a solution very similar to the way the version history in LINCKS is implemented to represent the forward linking, see appendix A. However this approach had a number of potential problems such as redundant storage of information, very big nodes, centralized storage and it did not handle the backward links. My second attempt used the backward approach to handle the command history, this is also the solution used in the implementation.

The backward links can be stored as proper links in the object versions attached to their root ends. This could not be done with the forward links, since we do not know which commands will follow the command that had just been executed (at least it can not be done in an easy way in the LINCKS system). The distributed storage of the backward link has a direct advantage over the centralized storage of the forward links in that the information of the preceding database commands is stored in the corresponding command node and not somewhere else. Another advantage is that the size of the nodes that have to be retrieved to calculate the preceding database commands is small.

Since we most often are concerned only with the most recent end of the command history, the distributed storage gives a gain in information transfer time between database and workspace.

The backward linking also has the important advantage that it will contain a representation of the forward links by the way the command history is constructed. This is explained in more detail later.

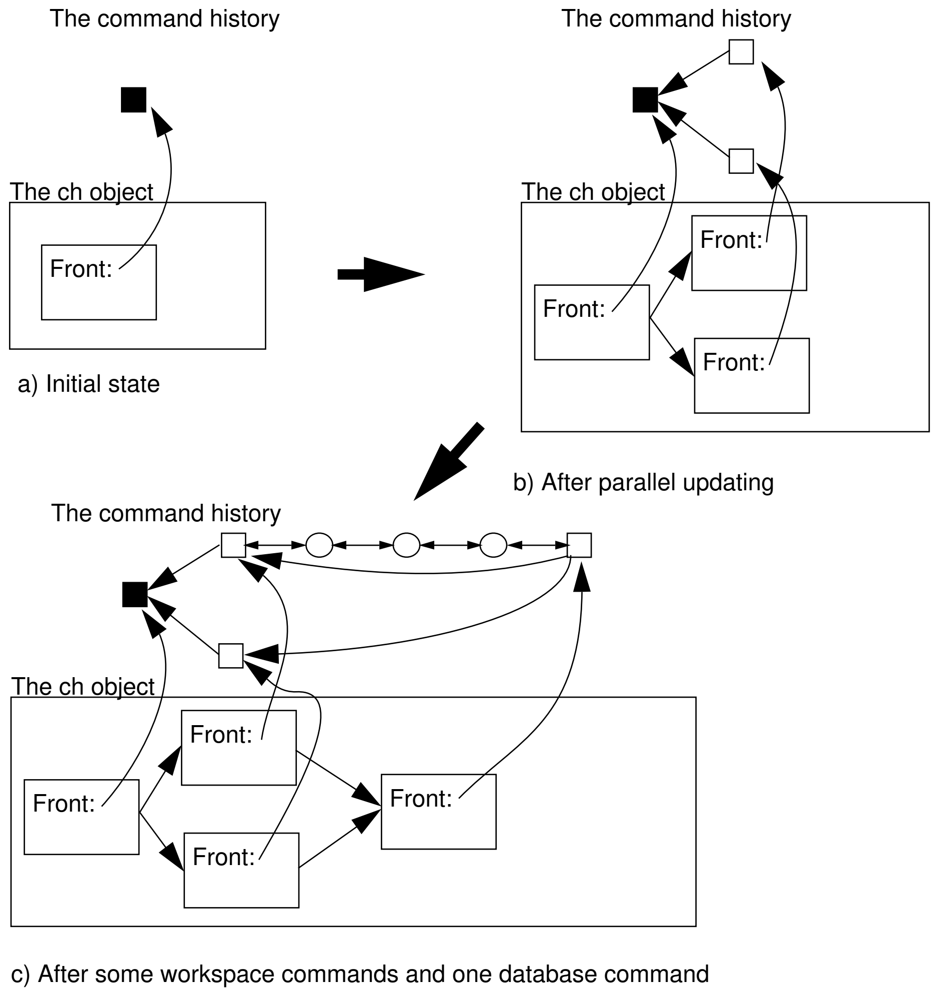
For the backward linking approach, the update of the command history is done according to the following rule.

- For each command (workspace and database): add a new version to the session history object and fill in the appropriate values.
- If it is a workspace command, then store the command node in the database and end the update.
- Otherwise add a new link group CH:Previous command to the command node. In this link group we store links to the previous database commands, i.e. the ones that were front commands when we started to update the command history. These are found by traversing the latest of the ch object and retrieving the 'Front' link in each of the latest versions.
- Next we create a new version of the ch object and store a link 'Front' that points to the new command node. The latest version of ch object is set to be this new ch object version.
- Finally, store the new ch object version.

It is however important to note that the succession relation is indirectly available in the temporal development of the ch object. We illustrate this in figure 14, which corresponds to the following chain of events. At first, there are no commands in the command history. This is represented as a special null-command node (the null-command node is used because we want to have exactly one start node in the command history, it also simplifies the update, since the initial case is included in the normal update case). Then one user executes a database command (e.g. a login). To update the command history, he retrieves the then current ch object (figure 14 (a)). Before he stores the updated ch object another user executes a database command and therefore retrieves the same (still current) version of the ch object. After this, the first user stores his new ch object and the second user finish his updating and stores his new ch object in the database. We have then the situation showed in figure 14 (b), where the ch object has parallel versions. The parallel versions of the ch object corresponds directly to the parallel occurrence of database commands. Continuing the example, the first user executes a sequence of workspace commands, and then another database command, which results in a command history as in figure 14 (c).

As we see in figure 14, the temporal history of the ch object (available in the VSNET group of the history structure node for the ch object) always reflects the history of database commands. To be able to use it for traversing the command history in succession order, we add a back link for each Front link, as illustrated in figure 15. The back link has not been implemented, since the management of the temporal history is currently being changed. When the back link is added it might be called 'CH:succ command' to reflect that it contains (indirectly) links to succeeding database commands.

In order to find the database commands that succeed a database command X we follow the back link from X to the corresponding ch object version. We then find succeeding versions from the ch objects history structure node and from them follow each front link to find the actual commands.



The arcs in the command history are stored in the nodes of the command history. The arcs from the versions of the ch object to the command history are the front links. The arcs in the ch object, between the versions, are stored in the history structure node of the ch object.

Figure 14: An example of how the command history evolves.

7.2.5 Pros and cons of the representations

The two different approaches to represent the command history, i.e. backward versus forward linking, give some different advantages. To make the decision easier I have summoned them here:

- The implementation of the backward representation gives about 30% less code than the forward representation. The code that is removed is concerned with the merge of parallel versions of the ch object. This gives simpler, faster update and more reliable code.
- In the forward representation we ended up with redundant storage of the command history. This was avoided in the backward case.

In both the forward and the backward representation we end up with very big history structure nodes. This is more of a system design matter, but it may cause

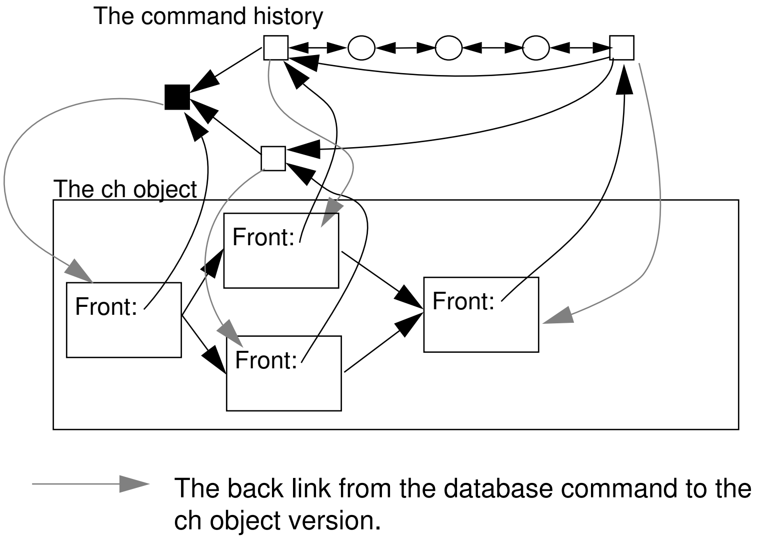


Figure 15: Connections command history - ch object

problems in the future with long response times.

It is important, for the decision on which representation to use, to note that the forward representation can be improved to avoid the redundant storage in the ch object version. This is done by simply removing the CHNET:after link group. By this we get the same structure of the command history as with the backward representation, see figure 14 and figure 15, except that we have not stored the precedence relation.

All things considered, the decision of which approach to take is quite simple: since the (improved) forward representation is included in the backward representation, we use the backward representation.

7.3 Logged Commands

???? Appendix C and possible D ...

7.4 Functions without Command History

The following functions prefixed with **CH_** corresponds to the same function, but the **CH_** will *not* call the comand history function. Notice if the **DBS** and **LIBLINCKS** are installed without the command history, then prefixed and non-prefixed function are equivalent. ???? Use something like Destructive for it?

Those marked with '*' have an extra parameter ....

```
errcode CH_CO_CREATEOBJ(lbl)
errcode CH_GA_GETATTR(lbl, group, field, val)
errcode CH_GLI_GETLINKITEM(lbl, group, field, pos, item, type)  *
errcode CH_GLN_GETLINKNAMES(lbl, group, mapfn, extra, type) *
```

```
errcode CH_GLV_GETLINKVAL(lbl, group, name, mapfn, extra)
errcode CH_GL_GETLATEST(lbl, mapfn, extra, inws) *
errcode CH_LINK(from, group, field, to, pos) *
errcode CH_RAT_REMOVEATTRTAG(lbl, group, field)
errcode CH_RFL_REMOVEFROMLATEST(lbl)
errcode CH_RLT_REMOVELINKTAG(lbl, group, field)
errcode CH_RLV_REMOVELINKVAL(lbl, group, field)
errcode CH_SA_SETATTR(lbl, group, field, val)
errcode CH_SO_STOREOBJ(lbl)
```

8 LIBLINCKS Functions

General policy: functions return zero for normal return, a positive integer for argument errors, and a negative integer for global errors (such as that the data base is not connected). See section 10, Error Codes. Exceptions to this rule are the status checking functions (**INWORKSPACE()**, **EDITED()**, **ISTRANSIENT()** and **BOUND()**) which return *TRUE* or *FALSE*.

Functions that are marked *[DESTRUCTIVE]* have the side effect that the input label is set to refer directly to the version that contains changes. That version is thereafter changed destructively until it is stored in the data base through a call to **SO_STOREOBJ()**.

8.1 Functions on the Attribute Section

CAGN_CHANGEATTRGROUPNAME(lbl,from,to)	*[DESTRUCTIVE]*
CAN_CHANGEATTRNAME(lbl,group,from,to)	*[DESTRUCTIVE]*
GAGN_GETATTRGROUPNAMES(lbl,mapfn,extra)	
GAN_GETATTRNAMES(lbl,group,mapfn,extra)	
GA_GETATTR(lbl,group,field,val)	
RAGT_REMOVEATTRGROUPTAG(lbl,group)	*[DESTRUCTIVE]*
RAGV_REMOVEATTRGROUPVAL(lbl,group)	*[DESTRUCTIVE]*
RAT_REMOVEATTRTAG(lbl,group,field)	*[DESTRUCTIVE]*
RAV_REMOVEATTRVAL(lbl,group,field)	*[DESTRUCTIVE]*
SA_SETATTR(lbl,group,field,val)	*[DESTRUCTIVE]*

8.2 Functions On The Links Section

CLGN_CHANGELINKGROUPNAME(lbl,from,to)	*[DESTRUCTIVE]*
CLN_CHANGELINKNAME(lbl,group,from,to)	
GLGN_GETLINKGROUPNAMES(lbl,mapfn,extra)	
GLI_GETLINKITEM(lbl,group,field,pos,item)	
GLN_GETLINKNAMES(lbl,group,mapfn,extra)	
GLV_GETLINKVAL(lbl,group,name,mapfn,extra)	
LINK(from,group,field,to,pos)	*[DESTRUCTIVE]*
RLGT_REMOVELINKGROUPTAG(lbl,group)	*[DESTRUCTIVE]*
RLGV_REMOVELINKGROUPVAL(lbl,group)	*[DESTRUCTIVE]*
RLT_REMOVELINKTAG(lbl,group,field)	*[DESTRUCTIVE]*
RLV_REMOVELINKVAL(lbl,group,field)	*[DESTRUCTIVE]*
ULA_UNLINKAT(from,group,field,pos)	*[DESTRUCTIVE]*
UL_UNLINK(from,group,field,to)	*[DESTRUCTIVE]*

8.3 Functions On The Image Section

GI_GETIMAGE(lbl,val)
SI_SETIMAGE(lbl,val) *[DESTRUCTIVE]*

8.4 Administration Functions

CO_CREATEOBJ(lbl)
CPO_COPYOBJ(obj,src) *[DESTRUCTIVE]*

GSR_GETSYSTEMROOT(lbl)

GC_GETCURRENT(lbl)
SC_SETCURRENT(lbl)
UBL_UNBINDLABEL(lbl)

GVS_GETVERSIONS(lbl,mapfn,extra)
GL_GETLATEST(lbl,mapfn,extra)
RFL_REMOVEFROMLATEST(lbl)

RO_RELEASEOBJ(lbl)
SO_STOREOBJ(lbl)

INWORKSPACE(lbl)
ISTRANSIENT(lbl)
EDITED(lbl)
BOUND(lbl)

LOGIN(dbid)
NOPROMPT_LOGIN(user,passwd,dbid)
LOGOUT()

RCB_REGISTERCALLBACK(what,callback,extra)
PEN_RETRIEVE_MESSAGE(mapfn,extra)

8.5 Functions on The Binding Tables

OS_OBJECTSYNC(lbl)
GBT_GETBINDINGTABLE(gpd,root,bt,created)
GBTC_GETBTCOMP(bt,lbl,changeable,part)

8.6 Functions on The Command History

CH_bad_lbl(lbl)
CH_init(app)
CH_update(10 Parameters)
CH_set_application(app)

8.7 Example

```
/*
 * This code demonstrates the use of some liblincks routines,
 * in particular the use of "mapfn" (map functions).
 * It can be incorporated in the xaim module aimcommand.c,
 * and invoked in one of the button (command) functions
 * like this:
 *
 *      if (currentinfonode == NULL)
 *              printf("no current item\n");
 *      else
 *              demo_liblincks(&currentinfonode->obj);
 *
 * To use lint on this file:
 *      lint -I/usr/local/X11R5/include xxx.c
 * because some X stuff is included via nested .h's.
 */

#include        "aimtypes.h"

static int      mapfn_group_name();
static int      mapfn_field_name();

/* This struct is for passing multiple parameters to a
 * mapfn with one pointer. */
struct demo_extra
        {
        label   *Plabel;
        char    *Pgroup_tag;
        };
```

```
/************************************************************
 * Function: static void demo_liblincks(label *Plabel)
 *
 * INPUT: a pointer to a node label.
 * OUTPUT: printfs of the nodes's image, attribute tags,
 *      values, etc.
 */
static void demo_liblincks(Plabel)
  label   *Plabel;
{
  attrval attr;

  if (GI_GETIMAGE(Plabel,&attr) != NO_ERROR) {
    printf("GI_ failed\n");
    return;
  }
  printf("Image of selected node:\n%s\n", attr.attvalue);

/*
 * liblincks allocated some memory, which is pointed to by
 * attr.attvalue.
 * We have the responsibility for freeing that memory when
 * we're finished with it.
 */
  free(attr.attvalue);

  /* we use the "extra" parameter to pass Plabel to the mapfn */
  if (GAGN_GETATTRGROUPNAMES(Plabel,mapfn_group_name,
                             (char *) Plabel) != NO_ERROR) {
    printf("GAGN_ failed\n");
    return;
  }
} /* end demo_liblincks() */
```

```
/***********************************************************
 * Function: static int  mapfn_group_name(FOUR PARAMETERS)
 * Parameters:
 *   label   *Plabel
 *   char    *Pgroup_tag;
 *   int     count
 *   int     length
 *
 * mapfn for a group.
 * Will be called by GAGN_ once for "initialization", then once
 * for each attribute group.
 */
static int mapfn_group_name(Plabel,Pgroup_tag,count,length)
  label   *Plabel;        /* using the "extra" parameter */
  char    *Pgroup_tag;
  int     count;
  int     length;
{
  struct demo_extra demo_parms;

  if (count == 0) {
  /* this is the first time we've been called for this node
   * ("initialization") */
    printf("This node has %d group(s).\n", length);
  }
  else {
    /* we're being called for a particular group */
    printf("\tGroup %d group tag: %s\n", count, Pgroup_tag);

    /* we store some information in a special block, so it
    * can be passed to the map function */
    demo_parms.Plabel = Plabel;
    demo_parms.Pgroup_tag = Pgroup_tag;

    if (GAN_GETATTRNAMES(Plabel,Pgroup_tag,mapfn_field_name,
                         (char *) &demo_parms) != NO_ERROR) {
      printf("GAN_ failed\n");
      return (!0);    /* what GAGN_ wants for failure */
    }
  }
  return (0);     /* this is what GAGN_ wants */
} /* end mapfn_group_name() */
```

```
/************************************************************
 * Function: static int mapfn_field_name(FOUR PARAMETERS)
 * Parameters:
 *    struct demo_extra *Pdemo
 *    char *Pfield_tag
 *    int count
 *    int length
 *
 * mapfn for a field.
 * Will be called by GAN_ once for "initialization", then once
 * for each attribute field within a group.
 */
static int mapfn_field_name(Pdemo,Pfield_tag,count,length)
  struct demo_extra *Pdemo;
  char *Pfield_tag;
  int count;
  int length;
{
  attrval attr;

  if (count == 0) {
     /* this is the first time we've been called for this group
     * ("initialization") */
     printf("\tThis group has %d field(s).\n", length);
  }
  else {
    /* we're being called for a particular field */
    printf("\t\tField %d field tag: %s\n", count, Pfield_tag);

    if (GA_GETATTR(Pdemo->Plabel,Pdemo->Pgroup_tag,
                   Pfield_tag,&attr) != NO_ERROR) {
      printf("GA_ failed\n");
      return (!0);    /* what GAN_ wants for failure */
    }
    printf("\t\tField value: %s\n", attr.attvalue);

    /* liblincks allocated some memory, which is pointed to by
    * attr.attvalue.
    * We have the responsibility for freeing that memory when
    * we're finished with it.
    */
    free(attr.attvalue);
  }

  return (0);     /* this is what GAN_ wants */
} /* end mapfn_field_name() */
```

9 Function Definitions

9.1 Functions on the Attribute Section

9.1.1 CAGN_CHANGEATTRGROUPNAME(lbl,from,to)

```
label *lbl;
char *from;
char *to;
```

[DESTRUCTIVE]

Changes an attribute group tag. Duplicate tags are not checked.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such 'from' group.
ERR_TAG	(3)	'to' group name too long.

9.1.2 CAN_CHANGEATTRNAME(lbl,group,from,to)

```
label *lbl;
char *group;
char *from;
char *to;
```

[DESTRUCTIVE]

Changes an attribute field tag. Duplicate tags are not checked.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.
ERR_TAG	(3)	Too long tag.
ERR_FIELD	(4)	No such field.

9.1.3 GAGN_GETATTRGROUPNAMES(lbl,mapfn,extra)

label *lbl;
int (*mapfn)();
char *extra;

Calls `(*mapfn)` once for each group in the attribute section of the given version. The call format is:

`(*mapfn)(extra,tag,count,length);`

where the arguments have the following declarations and contents:

- *char *extra*

  is passed in to the **LIBLINCKS** function as an extra application program parameter. It is not used within **LIBLINCKS**, and is intended as a pointer to a call block for application program arguments to the map function.

- *char *tag*

  points to the tag text as a null-terminated string placed in the internal group tag buffer.

- *int count*

  contains the sequence number of the current group. The first group has number one (1), the second has number two (2), and so on. This enumeration is done every time **GAGN_GETATTRGROUPNAMES** is called.

- *int length*

  contains the number of groups in the attribute section. This number stays the same over all calls to the map function.

The map function is first called with a NULL 'tag', and a zero 'count'. This call is intended for initialisation. The mapping continues until the map function returns non zero, or there are no more groups.

See section 8.7 for an example of using a *mapfn*.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.1.4 GAN_GETATTRNAMES(lbl,group,mapfn,extra)

label *lbl;
char *group;
int mapfn();
char *extra;

Calls `(*mapfn)` once for each field in the specified group in the attribute section of the given version. The call format is:

`(*mapfn)(extra,tag,count,length);`

where the arguments have the following declarations and contents:

- *char *extra*

  is passed in to the **LIBLINCKS** function as an extra application program parameter. It is not used within **LIBLINCKS**, and is intended as a pointer to a call block for application program arguments to the map function.

- *char *tag*

  points to the tag text as a null-terminated string placed in the internal field tag buffer.

- *int count*

  contains the sequence number of the current field. The first field has number one (1), the second has number two (2), and so on. This enumeration is done every time **GAN_GETATTRNAMES** is called.

- *int length*

  contains the number of fields in the given group in the attribute section. This number stays the same over all calls to the map function.

The map function is first called with a NULL 'tag', and a zero 'count'. This call is intended for initialisation. The mapping continues until the map function returns non zero, or there are no more fields.

See section 8.7 for an example of using a *mapfn*.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.

9.1.5 GA_GETATTR(lbl,group,field,val)

> label *lbl;
> char *group;
> char *field;
> attrval *val;

Retrieves the value of the given field in the given group in the attribute section. *malloc*'s a memory block, and copies the value into it. Sets '`val->attvalue`' to point to the memory block. The application program is responsible for *free*'ing the memory block.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.
ERR_FIELD	(4)	No such field.

9.1.6 RAGT_REMOVEATTRGROUPTAG(lbl,group)

label *lbl;
char *group;

[DESTRUCTIVE]

Removes a group from the attribute section, i.e. removes the specified group tag, and all field tags under that group tag, and their values.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.

9.1.7 RAGV_REMOVEATTRGROUPVAL(lbl,group)

```
label *lbl;
char *group;
```

[DESTRUCTIVE]

Removes all fields and their values from an attribute group. The group tag remains.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.

9.1.8 RAT_REMOVEATTRTAG(lbl,group,field)

label *lbl;
char *group;
char *field;

[DESTRUCTIVE]

Removes an attribute field, i.e. both field tag and value. The group tag remains.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.
ERR_FIELD	(4)	No such field.

9.1.9 RAV_REMOVEATTRVAL(lbl,group,field)

label *lbl;
char *group;
char *field;

[DESTRUCTIVE]

Removes the value of an attribute field. The field tag remains, and the value becomes a zero length block.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.
ERR_FIELD	(4)	No such field.

9.1.10 SA_SETATTR(lbl,group,field,val)

label *lbl;
char *group;
char *field;
attrval *val;

[DESTRUCTIVE]

Assigns a new value for an attribute field by copying the contents of the memory block referred to by '*val'. The input memory block (at `val->attvalue`) is not *free*'ed.

The given group and field are created if they don't exist beforehand.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.
ERR_FIELD	(4)	No such field.

9.2 Functions on the Links Section

9.2.1 CLGN_CHANGELINKGROUPNAME(lbl,from,to)

```
label *lbl;
char *from;
char *to;
```

[DESTRUCTIVE]

Changes a link group tag. Duplicate tags are not checked.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.
ERR_TAG	(3)	Too long tag.

9.2.2 CLN_CHANGELINKNAME(lbl,group,from,to)

label *lbl;
char *group;
char *from;
char *to;

Changes a link field tag. Duplicate tags are not checked.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.
ERR_TAG	(3)	Too long tag.
ERR_FIELD	(4)	No such field.

9.2.3 GLGN_GETLINKGROUPNAMES(lbl,mapfn,extra)

label *lbl;
int (*mapfn)();
char *extra;

Calls `(*mapfn)` once for each group in the link section of the given version. The call format is:

`(*mapfn)(extra,tag,count,length);`

where the arguments have the following declarations and contents:

- *char *extra*

  is passed in to the **LIBLINCKS** function as an extra application program parameter. It is not used within **LIBLINCKS**, and is intended as a pointer to a call block for application program arguments to the map function.

- *char *tag*

  points to the tag text as a null-terminated string placed in the internal group tag buffer.

- *int count*

  contains the sequence number of the current group. The first group has number one (1), the second has number two (2), and so on. This enumeration is done every time **GLGN_GETLINKGROUPNAMES** is called.

- *int length*

  contains the number of groups in the link section. This number stays the same over all calls to the map function.

The map function is first called with a NULL 'tag', and a zero 'count'. This call is intended for initialisation. The mapping continues until the map function returns non zero, or there are no more groups.

See section 8.7 for an example of using a *mapfn*.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.2.4 GLI_GETLINKITEM(lbl,group,field,pos,item)

```
label *lbl;
char *group;
char *field;
int pos;
label *item;
```

Assigns to '*item' the value of a given entry in a link field value. If 'pos' is less than one (1), the first entry is taken, and if 'pos' is greater than the number of items, the last entry is taken. However, these cases are reported with ERR_POS (5).

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.
ERR_FIELD	(4)	No such field.
ERR_POS	(5)	No such position.

9.2.5 GLN_GETLINKNAMES(lbl,group,mapfn,extra)

label *lbl;
char *group;
int mapfn();
char *extra;

Calls (*mapfn) once for each field in the specified group in the link section of the given version. The call format is:

```
(*mapfn)(extra,tag,count,length);
```

where the arguments have the following declarations and contents:

- *char *extra*

  is passed in to the **LIBLINCKS** function as an extra application program parameter. It is not used within **LIBLINCKS**, and is intended as a pointer to a call block for application program arguments to the map function.

- *char *tag*

  points to the tag text as a null-terminated string placed in the internal field tag buffer.

- *int count*

  contains the sequence number of the current field. The first field has number one (1), the second has number two (2), and so on. This enumeration is done every time **GLN_GETLINKNAMES()** is called.

- *int length*

  contains the number of fields in the given group in the link section. This number stays the same over all calls to the map function.

The map function is first called with a NULL 'tag', and a zero 'count'. This call is intended for initialisation. The mapping continues until the map function returns non zero, or there are no more fields.

See section 8.7 for an example of using a *mapfn*.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.

9.2.6 GLV_GETLINKVAL(lbl,group,name,mapfn,extra)

label *lbl;
char *group;
char *name;
int mapfn();
char *extra;

Calls (*mapfn) once for each entry in the link value of the given field in the specified group in the link section of the given version. The call format is:

```
(*mapfn)(extra,item,count,length);
```

where the arguments have the following declarations and contents:

- *char *extra*

  is passed in to the **LIBLINCKS** function as an extra application program parameter. It is not used within **LIBLINCKS**, and is intended as a pointer to a call block for application program arguments to the map function.

- *label *item*

  points to the label placed in the internal label buffer.

- *int count*

  contains the sequence number of the current entry. The first entry has number one (1), the second has number two (2), and so on. This enumeration is done every time **GLV_GETLINKVAL()** is called.

- *int length*

  contains the number of entries in the link value. This number stays the same over all calls to the map function.

The map function is first called with a NULL 'item', and a zero 'count'. This call is intended for initialisation. The mapping continues until the map function returns non zero, or there are no more fields.

See section 8.7 for an example of using a *mapfn*.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.
ERR_FIELD	(4)	No such field.

9.2.7 LINK(from,group,field,to,pos)

```
label *from;
char *group;
char *field;
label *to;
int pos;
```

[DESTRUCTIVE]

Adds a link from the 'from' node to the 'to' node by inserting it at the given position. Thus, the new link is at position 'pos', and its previous contents is at position 'pos'+1 etc. If 'pos' is less than one (1), or greater than the number of entries plus one, the function immediately returns ERR_POS (5).

If 'pos' is one (1), the given group and field are created unless they exist beforehand.

Note: LINK creates a new version of *from, so therefore it binds *from. "*to" may be bound or unbound. You may want to UBL_UNBINDLABEL() "to" before you LINK to it.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_POS	(5)	No such position.

9.2.8 RLGT_REMOVELINKGROUPTAG(lbl,group)

```
label *lbl;
char *group;
```

[DESTRUCTIVE]

Removes a group from the link section, i.e. both the tag and all the fields.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.

9.2.9 RLGV_REMOVELINKGROUPVAL(lbl,group)

label *lbl;
char *group;

[DESTRUCTIVE]

Removes all fields from a link group. The group tag remains.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.

9.2.10 RLT_REMOVELINKTAG(lbl,group,field)

label *lbl;
char *group;
char *field;

[DESTRUCTIVE]

Removes a link field, i.e. both field tag and value.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.
ERR_FIELD	(4)	No such field.

9.2.11 RLV_REMOVELINKVAL(lbl,group,field)

label *lbl;
char *group;
char *field;

[DESTRUCTIVE]

Removes the value of a link field. The field tag remains.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.
ERR_FIELD	(4)	No such field.

9.2.12 ULA_UNLINKAT(lbl,group,field,pos)

```
label *lbl;
char *group;
char *field;
int pos;
```

[DESTRUCTIVE]

Removes an entry of a link value. The entry at 'pos'+1 is moved to be at 'pos' etc. If 'pos' is less than one (1), or greater than the number of entries, the function immediately returns ERR_POS (5).

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.
ERR_FIELD	(4)	No such field.
ERR_POS	(5)	No such position.

9.2.13 UL_UNLINK(lbl,group,field,to)

```
label *lbl;
char *group;
char *field;
label *to;
```

[DESTRUCTIVE]

Removes all links in a link field value that are links to the specified 'to' object. If 'to' refers to a particular version (a bound label), only links to that version are removed. Otherwise (an unbound label) all links to that object are removed.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.
ERR_FIELD	(4)	No such field.

9.3 Functions on the Image Section

9.3.1 GI_GETIMAGE(lbl,val)

label *lbl;
attrval *val;

Retrieves the image of a node. *malloc*'s a memory block, and copies the image into it. Sets '`val->attvalue`' to point to the memory block. The application program is responsible for *free*'ing the memory block.

Note that images are handled like attributes.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.3.2 SI_SETIMAGE(lbl,val)

label *lbl;
attrval *val;

[DESTRUCTIVE]

Assigns a new value to the image of a node by copying the contents of the memory block referred to by '*val'. The input memory block is not *free*'ed.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.4 Administration Functions

9.4.1 CO_CREATEOBJ(lbl)

label *lbl;

Returns a label to a new, "empty" object. (A history structure node and a version are created.) The new version of the object gets the attributes **SYSTEM:Created** and **SYSTEM:Owner** set to current date_time and current login user.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.

9.4.2 CPO_COPYOBJ(obj,src)

label *obj;
label *src;

[DESTRUCTIVE]

Copies the contents of 'src' to be a new transient version of 'obj'. This function sets the 'obj' label to refer to the new version of 'obj', while the 'src' label is left unaffected. Note that 'src' and 'obj' may refer to the same object. The effect is always that the 'src' version becomes a new version for 'obj'.

If 'src' is transient, then it is automatically stored prior to copying. If 'obj' is transient, then it also is stored prior to copying.

To create a new object as a copy of an existing, the new object must be created first through a call to **CO_CREATEOBJ()**. Thereafter the source version can be copied "into" the new object.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.4.3 GSR_GETSYSTEMROOT(lbl)

label *lbl;

Assigns to '*lbl' an unbound reference to the system root node, by setting vs to VS_SYS_ROOT and inst to INST_UNBOUND.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.4.4 GC_GETCURRENT(lbl)

label *lbl;

Assigns to '*lbl' a bound reference to the current version of the object it refers to.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.4.5 SC_SETCURRENT(lbl)

label *lbl;

Sets current according to '*lbl'. If the label is unbound, current is left unchanged.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.4.6 UBL_UNBINDLABEL(lbl)

label *lbl;

Assigns to '*lbl' an unbound reference to the node it refers to.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.4.7 GVS_GETVERSIONS(lbl,mapfn,extra)

label *lbl;
int (*mapfn) ();
char *extra;

Applies (*mapfn) to all items (versions) in the given object's history. For each item we have three slots and (*mapfn) is called on each slot in turn and given the corresponding labels. The slots (in call order) are:

1. the actual version of the object,
2. the command history [2] version for the store of the actual version (first item above),
3. the temporal predecessors of the actual version (first item above).

The call format is:

```
(*mapfn)(extra,item,count)
```

where the arguments have the following declarations and contents:

- *char *extra*

  is passed in to the **LIBLINCKS** function as an extra application program parameter. It is not used within **LIBLINCKS**, and is intended as a pointer to a call block for application program arguments to the map function.

- *label *item*

  points to a bound ???? label placed in the internal label buffer.

- *int count*

  corresponds to the current slots:

  - **GVS_MORE** A query for loading a new history fragment ???? Needs description of fragments etc.
  - **GVS_INIT** An initialization call.
  - **GVS_VERSION** The version call (beginning of slots).
  - **GVS_COMMAND** The store command label call.
  - **GVS_PREDECESSOR** The predecessor call.

Initialisation calls are done with a call to (*mapfn) with a NULL item and a zero 'count'. The mapping function can return:

- **GVS_STOP** to immediate terminate mapping and the call.
- **GVS_CONTINUE** when is GVS_MORE means *not* to load more fragments, otherwise to continue mapping.

[2] Of course, the command history have to be enable during installation.

- **GVS_NEXT** go to next history slot.

The store command label is omitted from the transient slots, but an extra case 2 call is inserted with a (label *)0 pointer; if the store command label values coincide with the version concerned, then the command label is in reality unknown and therefore the 2 call case is done with a (label *)0 pointer.

See section 8.7 for an example of using a *mapfn*.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.4.8 GL_GETLATEST(lbl,mapfn,extra)

label *lbl;
int (*mapfn)();
char *extra;

Calls (*mapfn) once for each member of the set of latest for the given object. The call format is:

```
(*mapfn)(extra,item,count,length);
```

where the arguments have the following declarations and contents:

- *char *extra*

  is passed in to the **LIBLINCKS** function as an extra application program parameter. It is not used within **LIBLINCKS**, and is intended as a pointer to a call block for application program arguments to the map function.

- *char *item*

  points to a bound label placed in the internal label buffer.

- *int count*

  contains the sequence number of the current member. The first member has number one (1), the second has number two (2), and so on. This enumeration is done every time **GL_GETLATEST()** is called.

- *int length*

  contains the total number of members in latest. This number stays the same over all calls to the map function.

The map function is first called with a NULL 'item', and a zero 'count'. This call is intended for initialisation. The mapping continues until the map function returns non zero, or there are no more members.

See section 8.7 for an example of using a *mapfn*.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.4.9 RFL_REMOVEFROMLATEST(lbl)

label *lbl;

Ensures that the version referred to by '*lbl' is not a member of the set of latest for that node.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.4.10 RO_RELEASEOBJ(lbl)

label *lbl;

Releases the workspace instance of the node referred to, so that it will be fetched from the data base at the next access. Release can be done in order to decrease the memory space occupied by the workspace, and to ensure that the node data is what is actually stored in the data base. A version not yet saved will be lost if it is released before it is stored.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.4.11 SO_STOREOBJ(lbl)

label *lbl;

Stores the workspace copy of an object version into the data base. A new version that is edited locally in the workspace, does not become available to other users until it is stored in the data base. After it is stored, subsequent changes of it again creates a new local version.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.4.12 INWORKSPACE(lbl)

label *lbl;

Checks whether a node is in the workspace. This can be useful for deciding whether to access certain information.

Function returns:

TRUE	(1)	Node in workspace
FALSE	(0)	Node not in workspace

9.4.13 ISTRANSIENT(lbl)

label *lbl;

Checks whether a node is a transient state.

Function returns:

TRUE	(1)	Node is transient
FALSE	(0)	Node is not transient

9.4.14 EDITED(lbl)

label *lbl;

Returns TRUE if EITHER the vs or the inst part of the label is editable. TRUE implies that MAYBE the node has been edited in the workspace but has not yet been stored in the database.

Function returns:

TRUE	(1)	either vs or inst is editable
FALSE	(0)	NEITHER vs NOR inst is editable

9.4.15 BOUND(lbl)

label *lbl;

Checks whether a label is bound to a particular version of an object or whether it is unbound.

Function returns:

TRUE	(1)	Label is bound
FALSE	(0)	Label is unbound

9.4.16 LOGIN(dbid)

char *dbid

Logs in to the **LINCKS** database system, prompting for user id and pasword on stdout/stdin, and starts up the connection to the database 'dbid'. See also NOPROMPT_LOGIN().

Login names and passwords should be set in the file passwd, by using the 'dbpasswd' utility. The passwd file has this format:

'login name' 'user id' 'password'

Each user should be on a separate line.

Function returns:

SUCCESS	(0)	Login completed successfully.
FAIL	(-1)	Probably the database not connected.
ERR_LOGIN	(6)	Incorrect password or user name.

9.4.17 NOPROMPT_LOGIN(user,passwd,dbid)

```
char *user; /* string with user name */
char *passwd; /* string with password */
char *dbid; /* string with database name */
```

Login user on specified database (dbid) with given userid and password, with no prompting. See also LOGIN().

Function returns:

SUCCESS	(0)	Login completed successfully.
FAIL	(-1)	Probably the database not connected.
ERR_LOGIN	(6)	Incorrect password or user name.

9.4.18 LOGOUT()

Logs out and breaks the connection to the database.

Function returns:

SUCCESS	(0)	Connection to database broken.
ERR_RCBAD	(-4)	Pending PEN in DBS. Retrieve and retry.

9.4.19 RCB_REGISTERCALLBACK(what,callback,extra)

int what;
int (*callback)(char *extra, int index, int atoob);
char *extra;

Register a callback functions which is called when the workspace send commands to DBS. When called with a NULL function or a NULL value, restores the default handler in **LIBLINCKS**. The call back function is declared to return to a integer value, even though it is not used yet. The 'extra' argument is passed to the callback function when it is called. The possible value for 'what' which correspond to different messages sent to DBS are:

- **RCB_EDIT** The specified function is called when a new version is created for an object (or a node ????).
- **RCB_STORE** The specified function is called when a node is stored.
- **RCB_RETRIEVE** The specified function is called when a workspace wants to retrieve a node.
- **RCB_RELEASE** The specified function is called when the workspace has released an object.
- **RCB_PEN** Register PEN function???? move information concerning here?
- **RCB_PEN_RESOLVE** Register PEN function???? move information concerning here?
- **RCB_PEN_CHECK** Register PEN function???? move information concerning here?
- **RCB_PEN_POLL** Register PEN function???? move information concerning here?

The call format of the callback function is:

`(*callback)(char *extra, int index, int moltype);`

where the argument have the following declaration and content:

- *char *extra* is passed in to the **LIBLINCKS** function as an extra application program parameter. It is not used within **LIBLINCKS**, and is intended as a pointer to a call block for application program arguments to the map function.
- *int index* is the molecule index to the object sent to the database (see section 3.3).
- *int moltype* Type of molecule specified in *index*, can be **VSS**, **VSE**, **VS**, **VSH**, **OTHER**, or **UNKNOWN_TYPE** defined in `dbcodes.h` ???? Should this be interface?

The function returns the previous callback function, if any.

9.4.20 PEN Callback function

void (*callback)(char *extra, int oob);

This function s called when the workspace receives a PEN (Parallel Editing Notification) message from DBS (or rather when we recieves an out of band data (OOB) on from the DBS). With a NULL value, restores the default handler in **LIBLINCKS**. The default call back function is installed in the library to handle the basic cases of notification described in section 9.4.21, used in the basic tools built using **LIBLINCKS**.

The format of the PEN callback function is:

`(*callback)(char *extra, int oob, int atoob);`

where the argument have the following declaration and content:

- *char *extra* is passed in to the **LIBLINCKS** function as an extra application program parameter. It is not used within **LIBLINCKS**, and is intended as a pointer to a call block for application program arguments to the map function.
- *int oob* is the character sent as out of band data, thus the type of PEN/OOB sent. Currently we only uses `OOB_MESSAGE_CODE` which is defined in `dbcodes.h`.
- *int atoob* which is extra information, zero(0) if we couldn't read the *OOB*, else it is one(1).

The callback function is called from a unix signal handler and should thus be careful of which functions it calls in turn. See the man page for 'signal(2)' for more information. It can set a flag which is checked in the application main loop or write a character on a pipe.

9.4.21 PEN_RETRIEVE_MESSAGE(mapfn,extra)

int (*mapfn)();
char *extra;

Retrieves PEN (Parallel Editing Notification) messages from DBS and passes them to the mapfn together with the extra argument. We ends the traversal (but we retrieve all messages from the dbs) when the function, `mapfn` returns a non zero value or we get to the end of the list.

The call format is:

`(*mapfn)(extra,index,type,msg,count,len);`

where the arguments have the following declarations and contents:

- *char *extra* is passed in to the **LIBLINCKS** function as an extra application program parameter. It is not used within **LIBLINCKS**, and is intended as a pointer to a call block for application program arguments to the map function.
- *int index* is the version structure molecule index to the object refered by the message (see section 3.3).
- *int type* is the type of pen message. It can be
  - `PEN_RESOLVE` when the history node given by `index` was merged from a parallel update.
  - `PEN_CHECK` the object given by `index` is being editing by user(s) specified in `msg`.
  - `PEN_POLL` user `msg` begin editing object `index`.
- *char *msg* is an extra paramter string allocated by **LIBLINCKS** and should be free by you. It contains (currently) a list of user(s) id.
- *int count* contains the sequence number of the current message. The first member has number one (1), the second has number two (2), and so on.
- *const int len* contains the total number of messages to retrieve. This number stays the same over all calls to the map function.

The map function is first called with a NULL `msg`, and a zero `count`, `index`, and `type`. This call is intended for initialisation. The mapping continues until the map function returns non zero, or there are no more messages.

See section 8.7 for an example of using a *mapfn*.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_STATUS		Protocoll problem - unknow error status.

9.4.22 RTL_REMOVETRANSIENTLINKS(lbl)

label *lbl;

RTL_REMOVETRANSIENTLINKS takes a bound label as input. The function searches for a field in the link group TH_TRANSIENT of the history structure node of 'lbl' that has 'lbl' as physically the last linkitem. If it finds such a field, it removes it. (The nodes that the field pointed to, remain however).

Function returns:

SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_TH_TRANSIENT	(8)	No TH_TRANSIENT last place link to lbl

9.5 Functions on Binding Tables

9.5.1 OS_OBJECTSYNC(lbl)

label *lbl;

The **OS_OBJECTSYNC** takes a version node as input and confirm with the DBS that the workspace copy of the object has an up-to-date history, e.g. resolves the version node with database version, if needed.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.

9.5.2 GBT_GETBINDINGTABLE(gpd,root,bt,created)

label *gpd;
label *root;
label *bt;
int *created;

Fetches the binding table for the gpd-root combination. We use the link value 'SYSTEM:Binding_Tables' in the root object, 'root', history node to hold an association list of gpd-bt pairs. The binding table for a given gpd-root combination is thus obtained by scanning the link value 'SYSTEM:Binding_Tables' for root, searching (every odd position) for the occurance of the gpd label, and picking the next label as the binding table.

If the search fails, a new object is created for binding table 'bt', and a gpd-bt association is made in the 'root' history node. In this case, the history node is also stored after the linking is made. The association is made using unbound 'gpd' and 'bt' labels.

The parameters are:

- *label *gpd* is a *bound* generic presentation description label.
- *label *root* is a *bound* root object label.
- *label *bt* is the resulting binding table.
- *int *created* which is TRUE if a new binding table where created else FALSE.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Label 'gpd' or label 'root' are unbound.
ERR_VERSION	(11)	Found the link, but no version is found.

9.5.3 GBTC_GETBTCOMP(bt,lbl,changeable,part)

label *bt;
label *lbl;
int changeable;
int part;

Bind ‘lbl’ using binding table ‘bt’ which must be bound. The ‘lbl’ history node is sought in ‘COMPOSITION:Root’, ‘COMPOSITION:Gpd’, or ‘COMPOSITION:Components’. If found, ‘lbl’ is bound to the binding table entry, else ‘lbl’ is bound using ‘bt’ and added to the binding table.

If ‘changeable’ is TRUE, the bound ‘lbl’ is then sought in its history node’s ‘SYSTEM:Latest’ to establish if there has been any update to the object after the ‘lbl’ version. If so, the binding table is updated to contain the new component version.

The possible values for ‘part’ are

- *0* to search in order for the binding history node in ‘COMPOSITION:Root’, ‘COMPOSITION:Gpd’, and ‘COMPOSITION:Components’ in ‘bt’.
- *1* to search for the binding history node in ‘COMPOSITION:Root’ in ‘bt’. ‘COMPOSITION:Components’ in ‘bt’.
- *2* to search for the binding history node in ‘COMPOSITION:Gpd’ in ‘bt’.

Function returns:

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label - ‘bt’ not bound.
ERR_VERSION	(11)	Object not found and ‘changeable’ not set.

9.6 Functions on The Command History

9.6.1 CH_bad_lbl(lbl)

label *lbl;

Check if the '*lbl' is not equal to either the command history node (ch_node) nor equal to the user process node. Return **SUCCES** if this is the case else **FAIL**. Notices a NULL pointer return **SUCCESS**, too.

SUCCESS	(0)	Nor command history nor user process node.
FAIL	(-1)	The other case.

9.6.2 CH_get_user_process_node(lbl)

label *lbl;

Returns in '*lbl' the user process node, e.g. the label to the next node where the next command will be stored. It is only used by **SO_STOREOBJ** to create a link from the created object to the creating command in the command history.

Function returns:

ERR_NODB	(-1)	Database not connected.
SUCCESS	(0)	Normal return.

9.6.3 CH_init(app)

char *app;

Set the name of the current application which is stored in each command node. Defaults to "Unspecified". The input string is copied and it is the callers responibility to free 'app'.

Initialize the command history. Creates the command history node (`ch_node`) and a null command node ('no command has been executed'). Make a link from the `ch_node's` first version to the null command node, which is named front. Finally, we create an unbound link in the system root's history structure node that points to the `ch_node`. This function should only be called once after connecting to the DBS.

Function returns (as well as the assorted "normal" error codes):

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.

9.6.4 CH_set_application(app)

char *app;

Set the name of the current application which is stored in each command node. The input string is copied and it is the callers responibility to free 'app'. The previous application name is returned.

9.6.5 CH_update(func,type,from_attr,to_attr, group,field,pos,lbl,from_lbl,to_lbl)

```
char *func;
int type;
char *from_attr, *to_attr, *group, *field;
int pos;
label *lbl, *from_lbl, *to_lbl;
```

Updates the command history with the commands named '*func'. Type is zero (0) for database commands and one (1) for workspace command. The remaining parameters corresponds to the parameters to command done (named in '*func').

Function returns (as well as the assorted "normal" error codes):

ERR_NODB	(-1)	Data base not connected.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	An incorrect label.

10 Error Codes

The following error codes can be returned by **LIBLINCKS** functions, and are defined in liblincks.h and sunlincks.h:

ERR_RCERROR	(-6)	DB: Error info sent
ERR_RCUUP	(-5)	DB: Unable to unpack mol
ERR_RCBAD	(-4)	DB: Bad operation
ERR_ALREADY	(-3)	DB: Already received
ERR_STATUS	(-2)	DB: Unknown status returned
ERR_NODB	(-1)	Data base not connected.
FAIL	(-1)	Failed return
NO_ERROR	(0)	Normal return.
SUCCESS	(0)	Normal return.
ERR_LABEL	(1)	Invalid label.
ERR_GROUP	(2)	No such group.
ERR_TAG	(3)	Too long tag
ERR_FIELD	(4)	No such field.
ERR_POS	(5)	No such link item.
ERR_LOGIN	(6)	Incorrect password or user name.
ERR_LASTLATEST	(7)	Last item in latest.
ERR_TH_TRANSIENT	(8)	No TH_TRANSIENT last place link to lbl
ERR_DOUBLE_EDITING	(9)	Two transient versions created at the same time in the same workspace
ERR_AMBIGUOUS	(10)	The binding of unbound label ambiguous.
ERR_VERSION	(11)	
ERR_RESOLVE	(12)	Resolve error - checknumber inconsistency.
ERR_PARAMS	(13)	An invalid parameters, likely null.

11 Caveats

11.1 Malloc Errors

Errors resulting from insufficient memory space are in general not caught. In particular it is assumed that *malloc* works. If such errors are to be dealt with it should be done by replacing *malloc* calls with calls to a special routine which calls *malloc*. If *malloc* fails it should be caught within this routine and space should be made by releasing objects within the workspace (using RO_RELEASEOBJECT). Then *malloc* should be called again. It is of course important not to release objects which are being used by the current user level function.

Objects which have not been edited and are not being used by the current function should be able to be released without any observable effect from a user standpoint.

11.2 Parameter Checking

Parameters passed into the user functions are assumed to be of correct number and type. This of course needs to be checked. The checking is assumed to be a part of the user interface.

LIBLINCKS functions do check such things as whether a label is valid, whether a position number given is out of range for a particular list, and where appropriate whether tag names exist. These problems are reported via an error code as the function return value.

11.3 Number of Molecules

The number of molecules able to be referenced at any one time is presently limited to ???? (some #def somewhere?). This is due to the interface table on the database being of a fixed length. This will be changed in future versions so that the table is reallocated as necessary, putting no limit on the number of references.