UCSF
UC San Francisco Electronic Theses and Dissertations

Title
Designing a modern molecular modeling environment

Permalink
https://escholarship.org/uc/item/3wr2g2b5

Author
Huang, Conrad Chung-Shih

Publication Date
1989

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Designing a Modern Molecular Modeling Environment

by

Conrad Chung-Shih Huang

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Medical Information Sciences

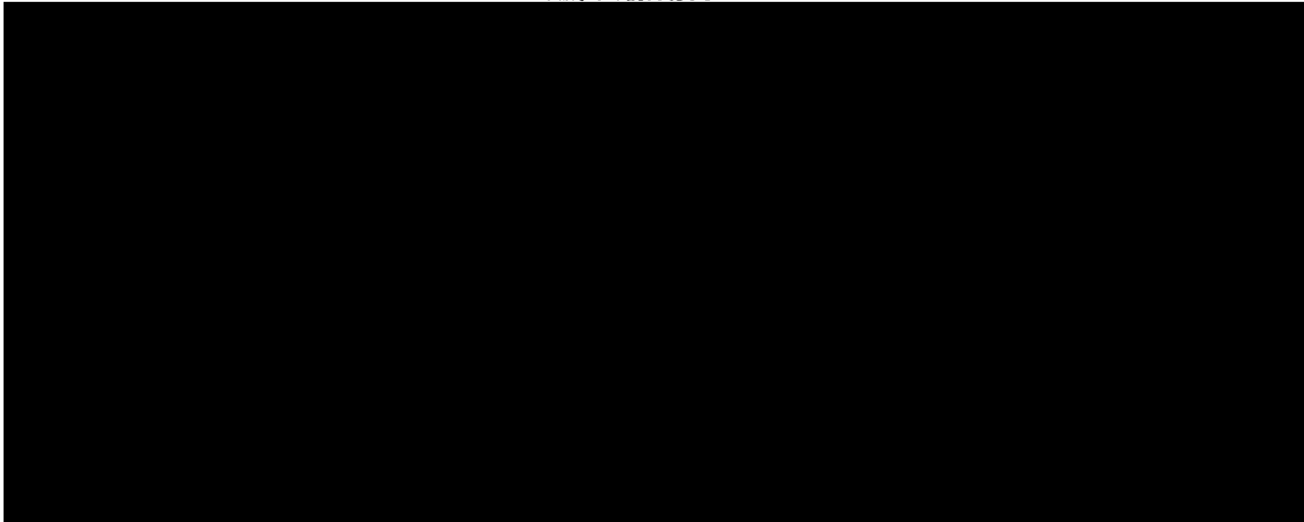
in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA

San Francisco



Date . . . . University Librarian

Degree Conferred: March 26, 1989

''It isn't that we build such bad cars, it's that they're such bad customers.''

— Charles F. Kettering
Director of Research at GM from 1920 to 1947

''User error.''

— Conrad Huang

Acknowledgement

I would like to thank the members of the Computer Graphics Laboratory for their contributions to and criticisms of this project: Professor Ferrin for a well-managed laboratory; Eric Pettersen, Greg Couch, and Scooter Morris for their constructive criticisms; Teri Klein, Don Kneller, Bruce Cohen, and Dave Spellmeyer for broadening my scientific horizons.

I thank Tom Ferrin and Bob Langridge for supporting me all these years; and my parents, brother, and Teri Klein who nagged until I gave in and finished this dissertation.

I thank my research committee: Professors Robert Langridge, I.D. (Tack) Kuntz, and Thomas Ferrin for their instruction and advice, and for serving on both my advancement to candidacy and thesis committee. In particular, Professor Langridge for guidance on what to do and how to do it; Professor Kuntz for suggestions on research directions; and Professor Ferrin for sharing his technical expertise.

Finally, I thank the NIH Division of Research Resources (RR-1081) and the DARPA University Research Initiative (N00014-86-K-0757) for providing the facilities and support.

Abstract

Computer modeling is an indispensable tool in studying molecular interactions. The goal of this project is to design and implement a macromolecular workbench (**MMWB**) which integrates the many techniques used in molecular modeling with modern computer hardware to create an easy-to-use and powerful modeling environment.

To achieve this goal, **MMWB** provides the following facilities: a single user interface, a single data communications protocol, and a single data management library. The user interface is a multi-layered system that facilitates user customization and extension. The data communications protocol is built on top of standard network protocol to transparently provide distributed computing. The data management library uses an object paradigm on top of tables to structure data hierarchically. Each facility provides a standard system service which simplifies the task of its users.

With standard system services, users need only learn a single system to access the entire range of applications in the environment; applications can communicate with each other without explicit prior arrangements; and data duplication is minimized since the same data may be shared by many applications. These characteristics make **MMWB** an effective and useful modeling tool.

Table of Contents

1. Designing a Modern Molecular Modeling Environment 1
- 1.1. Introduction 1
- 1.2. Design Goals 2
- 1.3. Implementation 3
  - 1.3.1. User Interface 4
  - 1.3.2. Data Communications Protocol 5
  - 1.3.3. Data Management 6
- 1.4. Summary 8

2. Network Communications of MMWB 8
- 2.1. Introduction 9
- 2.2. Establishing Network Connections 10
  - 2.2.1. Dispatcher 10
  - 2.2.2. Server 11
  - 2.2.3. Client 11
  - 2.2.4. Implementation 12
- 2.3. Exchanging Information 12
  - 2.3.1. Commands 13
  - 2.3.2. Responses 15
  - 2.3.3. Examples of Information Exchange 17
- 2.4. Summary 19

3. Application Services of MMWB 19
- 3.1. Introduction 20
- 3.2. Data Management 20
  - 3.2.1. Data Organization 21
  - 3.2.2. Data Access Protocol 22
  - 3.2.3. Concurrency Control 23
  - 3.2.4. Crash Recovery 24
- 3.3. Computation 24
- 3.4. User Interface 26
- 3.5. Summary 27

4. Data Management in MMWB 29
- 4.1. Introduction 29
- 4.2. Data Model 30

4.2.1. Items in the Data Model 32
4.2.2. Operations in the Data Model 33
4.2.2.1. Synchronization Operations 33
4.2.2.2. Data Description Operations 36
4.2.2.3. Data Manipulation Operations 37
4.2.2.4. Data Set Operations 38
4.3. Data Organization 39
4.3.1. Object Properties 40
4.3.2. Object Operations 40
4.3.3. Projects 41
4.4. Interface Specifications 41
4.4.1. Program Functional Interface 41
4.4.1.1. Data Types 42
4.4.1.2. Procedures 44
4.4.1.2.1. Data Manager Routines 45
4.4.1.2.2. Table Routines 46
4.4.1.2.3. Object Routines 49
4.4.2. Network Protocol Interface 51
4.5. Implementation Issues 59
4.5.1. Implementing the Data Manager 60
4.5.2. Implementing the Functional Interface 63
4.6. Summary 64
5. The Programming Framework for MIRAGE 64
5.1. Introduction 65
5.2. Design Approaches 65
5.2.1. Single-layer Design 65
5.2.2. Multi-layer Design 66
5.2.3. Hybrid Design 67
5.3. MIRAGE 68
5.3.1. System Primitives Layer 69
5.3.1.1. Molecular Primitives 70
5.3.1.2. External Primitives 70
5.3.1.3. Graphics Primitives 71
5.3.2. System Commands Layer 72
5.3.2.1. *meta*-language 72
5.3.3. Interpretation Layer 74
5.3.4. Summary of MIRAGE 75
5.4. Conclusion 75

6. Future Directions 77
  6.1. Introduction 78
  6.2. Unfinished Business 78
  6.3. Future Extensions 80
  6.4. Summary 80
References 81
Glossary 84
Appendix A. Dispatcher Interface 86
Appendix B. *meta*-language Grammar 89
Appendix C. Prototype Implementation 91

List of Tables

Table 1 – Service categories and design issues 27
Table 2 – Operations in the Data Model 38
Table 3 – Typical properties of objects. 40
Table 4 – Object operations 41
Table 5 – Functional specification variables and types 44
Table 6 – List of molecular primitives 70
Table 7 – List of external primitives 70
Table 8 – List of graphics primitives 72
Table 9 – Features of user interface programming frameworks 76

List of Figures

Figure 1 – Entity-relationship diagram for molecular data. 30
Figure 2 – Clients are cooperatively accessing table 35
Figure 3 – Internal structure of a file 61
Figure 4 – In-memory data structure of tables 61
Figure 5 – Objects implemented as tables 63
Figure 6 – *meta*-language description of *color* command 73

Chapter 1

Designing a Modern Molecular Modeling Environment

1.1. Introduction

Molecular modeling is an interdisciplinary field employing many different techniques to study interactions among biomolecules. One of the indispensable tools of molecular modeling is the computer, which is used for a wide range of tasks, including simulating molecular dynamics, computing minimum energy structures, exploring conformational space, and visualizing results. In view of the importance of the computer, existing molecular modeling software environments are surprisingly inadequate.

Currently, most molecular modeling software environments consist of several independently developed packages, each addressing a different area (*e.g.*, energy minimization [25], visualization [9], *etc.*). Each package has its own unique user interface and input format. The amount of time required to learn to operate each individual package is reasonably low; however, when there are many packages to learn, the startup time for users, time that may be spent doing research, becomes unacceptably lengthy. Additionally, each package requires maintenance by local administrators, who must learn many different programming conventions.

Another characteristic of existing software environments is that they tend to use only a single computer. With networking and workstations becoming the norm of computation, a distributed environment can take advantage of available hardware resources much more efficiently than the existing centralized ones.

The goal of this project is to design and implement a macromolecular workbench (MMWB) which integrates the many available packages and hardware so that users can

proceed straightforwardly to research, without spending much time in learning new systems and managing computing resources.

1.2. Design Goals

The major design goals of MMWB are

- correctness
- ease of maintenance
- ease of use
- ease of extension
- distributed computing

The first two goals can be achieved using standard software engineering practices. The quality of the software may be evaluated using the criteria of **modularity** and **integration**. Modularity measures how well a software system is partitioned. Integration measures how well the various partitions interact. Modularity ensures that components of the system may be maintained individually while integration ensures that the entire system operates cohesively as a whole.

The latter three goals may be achieved if we provide the following facilities within the environment.

- a single user interface to all computation services
- a single data communications protocol among programs
- a single data management library

A single user interface guarantees that users need only learn a single system for accessing the entire spectrum of services available in the environment. Of course, the interface must be self-consistent, to minimize learning time. It should take advantage of available hardware (*e.g.*, present graphical instead of textual information when appropriate). And it should be intuitive, to make the system easy to use.

A single communications protocol guarantees that programs share the same information exchange format. Individual programs will have different detailed formats, but they should be able to communicate with each other in generic ways (*e.g.*, request for parameters). Foreign programs may then be incorporated into the environment by writing a single conversion program, instead of modifying the foreign program or many existing programs. With a single protocol, programs within the environment can work cooperatively without explicit knowledge of each other. If the communications protocol is designed to be usable over a network protocol, then distributed computing naturally follows.

Finally, a single data management library guarantees that there is no proliferation of duplicated data. One of the main problems with the current environments is that the same data are stored several times in different formats for different programs. By having a single data management library, the duplication is reduced, and generic data management becomes possible.

1.3. Implementation

Designing the three necessary facilities requires detailed knowledge of several fields. To design a good user interface, one must understand the application domain, human factors [19], and presentation techniques [24]. To design a good communications protocol, one needs to have a good grasp of data communications issues [6, 23] and differences between the communicating systems. To design a good data manager, one needs to know the type of data to be stored and details of data storage techniques [7, 14]. Only by combining the expert knowledge of molecular modeling with the extensive algorithms of computer science can one formulate useful and usable facilities in **MMWB**.

1.3.1. User Interface

Some desirable properties of user interfaces are

- customizability, for ease of use
- extensibility, for flexibility
- consistency, for ease of learning

These properties may be obtained by creating a multi-layer user interface. The lowest layer implements the primitive operations supported by the interface, such as calling other services and changing display parameters. The middle layer implements a *meta-language* which is used to combine the primitive operations into user interface commands. The highest layer converts user actions into user interface commands.

Customizability is handled by the highest layer, which may implement several styles of interfaces. The user may select a preferred style (*e.g.*, menu-driven commands) with special options (*e.g.*, pull-down rather than pop-up menus). The specified style and options are used to determine how user actions, such as clicking a mouse button, are mapped to user interface commands.

Extensibility is handled by the middle layer, which implements all user interface commands. In most existing systems, a user action is mapped to a fixed series of program actions. The *meta-language* implemented by the middle layer provides an alternative. User actions are mapped to user interface commands, which consist of a series of primitive operations and are described in the *meta-language*. New commands may be easily created by combining primitive operations in new and different ways without ever modifying the user interface program.

Consistency is the most difficult of the three properties to achieve. If the user interface is easily customizable and extensible, consistency among users is nearly impossible.

A prime example is the many dialects of **LISP** that derived from pure **LISP** [26]. Instead of trying to achieve homogeneity of user interface styles, a more profitable approach is to provide concepts that exist across all interfaces, *i.e.*, use a *metaphor*. A metaphor is a mapping of user interface concepts to real world concepts. For example, the objects and actions in the interface may be modeled upon the objects on a chemistry workbench and the actions of the chemist. Creating metaphors is beyond the scope of this project. Instead, **MMWB** will rely on the discretion of users not to abuse the flexibility provided by the user interface.

1.3.2. Data Communications Protocol

In **MMWB**, programs executing on different computers must be able to communicate with each other over a network. The communications medium used by the programs should be reliable since message delivery is not an application concern. These constraints limit the possible choice of medium.

Three possible techniques for communicating over a network are via network file systems [22], remote procedure calls (RPC) [21,23], and transport layer messages [6, 16, 23]. Using network file systems is undesirable because synchronization of file operations is difficult. Remote procedure calls provide the ideal solution. Unfortunately, there is no standard remote procedure call software package. Furthermore, different RPC packages make different assumptions (*e.g.*, no multiple return value, different error handling), so there is no clear choice which RPC package is most suitable for the **MMWB** environment. Transport layer messages provide an acceptable compromise. Although there are several transport protocols being used, one is in widespread use in the academic community: the DARPA network protocol suite, including TCP/IP [6, 11].

Additionally, most of the transport layer protocols implement two message delivery methods: datagrams and streams. Thus, by designing a protocol on top of either datagrams or streams, one can easily migrate from one transport protocol to another.

The reliability requirement for the communications medium precludes datagrams because the DARPA suite does not support reliable datagrams. Thus the only choice remaining is using stream connections. This choice provides side benefits other than satisfying our requirements. Byte streams may be simulated on a single system easily (*e.g.*, UNIX[1] pipes and VMS[2] mailboxes). Also, streams are easy to monitor and debug, since output may be stored in a file and input may be read from a file.

The data communications protocol itself is a synchronous protocol that uses mostly text. The protocol is based on the client-server model of communications, where one side sends requests and the other sends responses to requests. Synchronous communications is much easier to implement than asynchronous ones because there is no need to handle unexpected input. The decision to use mostly text is made primarily for implementors, since textual protocols are easier to monitor and debug than binary ones. When all requests are textual, programmers can directly send commands to programs and view the output, thus facilitating debugging efforts.

1.3.3. Data Management

Data management in molecular modeling environments is becoming a very important issue. Several applications using commercial database systems for management have been attempted [17]. Although this approach is possible in principle, there are

[1] UNIX is a trademark of AT&T Bell Laboratories.

[2] VMS is a trademark of Digital Equipment Corporation.

several practical problems. First, most database packages are designed for businesses, where accounting and inventory are of primary importance. Thus, much of a database system is given to handling simultaneous transactions that modify data. Also, the packages are designed to handle a few very large databases rather than many small ones. Molecular data, on the other hand, tend to be static, modified only by a single user, and consist of many different classes of data, such as atomic data, molecular sequences, electron density maps, *etc*. Thus, the needs of molecular data management and the features of commercial database management systems do not match up well. A system designed specifically for molecular data would be simpler and faster than a commercial database system because it can take advantage of these properties of the data.

Data management in **MMWB** uses an object paradigm. Objects in the system include molecular data, user interface state, program descriptions, *etc*. Objects consist of properties, which include the object name, owner, access control list, and data. The data associated with an object are stored in tables. Tables in **MMWB** have named columns and ordered rows, thus they differ significantly from relational database principles [14]. The reason for not using relational algebra, with its rigorous mathematical basis, is simplicity, both for application programmers and **MMWB** implementors. Tables with ordered rows, where each row has a constant identifying row number, are easier to use in programs than relations. Furthermore, they are much easier to implement than relations.

By using an object system, two layers of organization are introduced at a level common to all programs. First is the division of data into objects, which must be accessed in a uniform way. There are no longer any ''floating'' data that must be accessed by some non-standard method. Second is the coercion of data into tables. Although tables are not

universally suitable for all applications, they are convenient as a **storage** format to be shared by all programs. Data from table, when read into main memory, may be reorganized in any fashion that is suitable for each applications. However, the storage format must be compatible among all applications. With these two levels of organization, ancillary programs may be written to manage data in a generic way without explicit knowledge about the semantic information contained in the data.

1.4. Summary

Computing, ranging from simulation to visualization, has become a central part of molecular modeling. Unfortunately, the existing software environments for modeling is inadequate due to the heterogeneity of packages and changing hardware environment. A new system is needed to reconcile the differences among the many packages and take advantage of newer technology such as computer networks.

The design target for the new system is a user-friendly, extensible, and distributed environment that integrates the many different tools available for molecular modeling.

Chapter 2

Network Communications of MMWB

2.1. Introduction

MMWB is designed to operate in a heterogeneous network environment to take advantage of the different capabilities of different hosts. Thus, a user may execute several processes on different hosts to achieve a single task. For example, when a user wants to monitor the progress of an energy minimization calculation, he may run a display program on a graphics workstation, the minimization computation on a supercomputer, and a data manager on a file server. The set of processes, display program, computation, and data manager, may be considered a single **conversation** between the user and **MMWB**. There may be many conversations for many users all executing simultaneously, so each is tagged with a unique **conversation identifier** to distinguish it from all others.

Processes within the same conversation should be able to communicate with each other, *e.g.*, minimization computation with data manager. Processes in separate conversations should be isolated from each other. The first networking requirement in **MMWB** is to provide a mechanism which enables processes in the same conversation to locate each other while preventing processes in different conversations from communicating accidentally. Process rendezvous is a problem in the session layer of the International Standard Organization Open System Interconnect (ISO/OSI) model of networking [16]. Once processes locate each other, they may exchange information. However, since **MMWB** works in a heterogeneous environment, the hosts that the processes run on may not be bitwise compatible (*e.g.*, one may use IEEE floating point format while another

uses a proprietary format). The hosts must agree on a mutually acceptable communications method before data can be exchanged. Data exchange format is a problem in the OSI presentation layer. Both these problems are discussed in detail in the following sections.

2.2. Establishing Network Connections

When two processes attempt to contact each other, their behavior is usually described by the **client-server** model. In this model, one process is the client which initiates the communications. The other process is the server which responds to the client's requests. A process in an MMWB conversation may be a client, a server, or both. In the example above, the display program is a client, the data manager is a server, and the energy minimization computation acts as the server to the display program and as the client to the data manager.

In the **client-server** model, the client is responsible for locating the server. To help clients in this difficult task, MMWB provides a **dispatcher** on each host to direct clients to server locations.

2.2.1. Dispatcher

The dispatcher is a program which executes on a single host and keeps track of what servers are currently running on that host.[3] It maintains a list of servers, along with server type, contact location, and conversation identifier. The dispatcher itself listens at a **well known location** so clients can easily contact it. When the dispatcher receives a query from a client, it waits for the client to present its conversation identifier and the desired

[3] The communications protocol used by the dispatcher is shown in Appendix A.

server type. The dispatcher then searches through its list of servers for one with matching server type and conversation identifier. If such a server exists, then the dispatcher sends the associated contact location back to the client. Otherwise, the dispatcher starts up a new server of the desired type, informs it that it is in the client conversation, assigns it a contact location, enters it into the dispatcher's list of servers, and sends the contact location back to the client. In either case, the client receives the contact location where the desired server may be found.

An additional task of the dispatcher is to keep track of which servers are currently active. When a server terminates, its associated entry must be removed from the dispatcher's list of servers, lest new clients be directed to contact a non-existent server. The dispatcher design is similar to the **ARPANET initial connection protocol** and **process server** [23].

2.2.2. Server

The task of the server is quite simple. A server is initiated by the dispatcher and given a conversation identifier. Its first task is to wait for a connection request from a client and verify that the client is using the same conversation identifier as the one that the dispatcher sent. Once this authentication is complete, the server may continue on to the task of providing its designed service.

2.2.3. Client

The client does most of the work in establishing a network connection between two processes. The first step that a client has to take is deciding what service it wants from which host. The client then contacts the dispatcher on that host at the **well known loca-**

tion of dispatchers. It presents its conversation identifier and desired service type. The dispatcher on the remote host should then return the contact location of a server. Using the returned contact location, the client can then reach the server, present its conversation identifier again, and send requests using a predefined information exchange format for that service.

2.2.4. Implementation

The previous sections discussed networking without referring to any underlying network protocols (*transport layer* in the OSI networking model). Any of several low-level protocols may be used, *e.g* TCP/IP, XNS [27], and ISO. The only requirement of the underlying protocol is that it permits a process on one host to contact a process on another host at a predetermined network location. MMWB is currently implemented in a TCP/IP environment where a **well known location** corresponds to a **well known service** and contact locations are TCP port numbers.

The dispatcher needs to be notified whenever a server it started terminates. In many systems, (*e.g.*, UNIX and VMS), a process is automatically notified by the operating system when a child process (one which is started by the original process) terminates. Thus, notification is simple in these systems. If this mechanism is not available, an alternate method, such as having the server contact the dispatcher prior to termination, must be devised.

2.3. Exchanging Information

Once a network connection is established between a client and a server, information may be exchanged. First, the server sends a greeting message, in the same format as a

response to a command; this is most useful for presenting version and functionality information. The client then continues the exchange by sending commands and waiting for further responses from the server. Because **MMWB** operates in a heterogeneous environment, these commands and responses should be passed in a host-independent format. The ideal solution is using some type of remote procedure call (RPC) mechanism. However, since there is no widely available standard RPC package, a simpler **ASCII**-based protocol built on top of the transport layer is used. Effectively, the simple-minded protocol is a very primitive RPC mechanism. **ASCII** is chosen because programmers can then converse directly with all server interfaces using a simple debugging program which relays keyboard input to the server and server responses to the debugging terminal.

The protocol used by **MMWB** is very similar to the **Network News Transfer Protocol** [10]. Commands are a command word followed by a parameter list. Responses are a three-digit numeric code followed by a short description of the response code. The following sections describe the set of commands and responses that all services must understand. Commands and responses which are specific to individual clients and servers are documented in the service description documents.

2.3.1. Commands

Commands are a series of white-space separated printable **ASCII** strings terminated by a newline character. The first string is the command word and all subsequent strings are parameters of the command. The following commands are reserved as generic commands.

VERS minimum maximum min max

VERS is the first command sent by a client to a server. The four

parameters are integers which represent the minimum and maximum protocol version numbers that the client understands. *minimum* and *maximum* refer to the generic initial handshaking that is exchanged (see the **HELO** and **USER** commands below). *min* and *max* refer to the specific service protocol version. The reply from the server consists of two numbers, specifying the handshaking protocol version and the service protocol version (in that order) that the client should use.

HELO conversation_id

HELO is the second command sent by a client to a server. The single parameter of this command is a conversation identifier of the form

project_id@host_id

where *project_id* is a series of printable characters, excluding '@', and *host_id* is a string compatible with the ARPA Internet host names [15]. The *host_id* uniquely identifies the host from all other hosts. Generally, the *host_id* is a fully qualified domain name that is registered with the ARPA Internet. The *project_id* uniquely identifies a group of cooperating clients from all other groups executing on host *host_id*. Generally, the *project_id* is a file name or some registered project name. The **HELO** command identifies the client to the server which determines access permissions based on the conversation identifier.

USER account requestor

USER is the third command sent by a client to a server. It is used by the server to determine whether the client has the necessary privileges to

receive service. The first argument, the account name to use on the server host, consists of a string of alphanumeric characters. The second argument, a description of the identity of the requestor, is of the form

account@host_id

where *account* and *host_id* have the same format as above. In the future, this command will be superceded by more sophisticated authentication mechanisms such as the Kerberos authentication system [20].

HELP **HELP** is the command sent by a client when it needs a list of available commands and short description of each one. More elaborate **HELP** schemes may be supported but are not required.

QUIT **QUIT** is the last command sent by a client to a server. This command notifies the server that the client will not send any more commands.

2.3.2. Responses

A response consists of an ASCII string of printable characters terminated by the newline character (hexadecimal 0xA) optionally followed by data of unspecified format. The ASCII string of characters is a general response line of the form

nnn response_text

where *nnn* is a response code consisting of three decimal digits, and *response_text* is a textual description of the response code. The *response_text* may also contain parameters describing any optional data which follow the general response line.

Each digit in the response code represents a general aspect of the response. The first digit represents the status of the command.

1xx	Informative message
2xx	Command accepted
3xx	Incomplete command accepted; waiting for further input
4xx	Correct command rejected for some reason
5xx	Incorrect command rejected
6xx	Last response to command

The second digit represents the type of response.

x0x	Response to unclassified request
x1x	Response to data retrieval request
x2x	Response to computation request
x8x	Response to non-standard extensions
x9x	Response containing only debugging information

The third digit represents command-specific response codes which are described with each command. The following codes are reserved as generic responses.

100	Help text
190 – 199	Debug output
200	*Generic acknowledgement message*
201	*Reserved for future use*
202	ASCII data follow
203	*n* octets of binary data follow
204	No data available
400	Server is quitting
401	Protocol version mismatch
402	Permission denied
403	Internal server error
500	Unknown command
501	Command syntax error
600	Last response (must be sent for each command)

The optional data which follow responses may be of two formats: ASCII or binary. ASCII data are sent as arbitrary strings terminated by the newline character. The length of each string, including the newline character, must not exceed 1023 characters. ASCII data are terminated by a string containing exactly one period (hexadecimal 0x2E) followed by a newline character. Binary data are sent in a single block of network-order

octets. The response text preceding the binary data must specify the number of octets in the binary data.

The initial greeting message that the server sends consists of a generic acknowledgement message (code 200) followed by the end of message (code 600).

2.3.3. Examples of Information Exchange

In the following examples, C stands for client and S stands for server.

```
S:	200 Data manager at your service
S:	600 End of response.
C:	VERS 1 2 1 1
S:	200 Protocol versions follow
S:	202 ASCII data follow
S:	1 1
S:	.
S:	600 End of response.
C:	HELO 12345@cgl.ucsf.edu
S:	200 Mirage data manager on host socrates.ucsf.edu ready.
S:	600 End of response.
C:	USER conrad conrad@cgl.ucsf.edu
S:	200 User okay.
S:	600 End of response.
C:	OPEN /usr/mol/midas/1gcn
S:	210 /usr/mol/midas/1gcn opened.
S:	600 End of response.
C:	OPEN /usr/mol/midas/3fxn
S:	410 /usr/mol/midas/3fxn locked.
S:	600 End of response.
C:	TOC /usr/mol/midas/1gcn
S:	211 Table of contents follows
S:	202 ASCII data follow
S:	atom
S:	residue
S:	model
S:	surface
S:	.
S:	600 End of response.
C:	QUIT
S:	400 Data manager closing connection.
S:	600 End of response.
```

An example of a less pleasant exchange is:

```
S:   200 Data manager at your service
S:   600 End of response.
C:   VERS 1 2 1 1
S:   200 Protocol versions follow
S:   202 ASCII data follow
S:   1 1
S:   .
S:   600 End of response.
C:   HELO 12345@cgl.ucsf.edu
S:   402 Permission denied, conversation_id mismatch
S:   400 Data manager closing connection.
S:   600 End of response.
```

2.4. Summary

Processes in MMWB can easily communicate with each other by using the protocols described in the previous sections. Conversation contexts are preserved by maintaining a conversation identifier for each group of processes that are part of a common task. Debugging is simplified by using a user-interpretable information exchange format such that all services may be accessed via a single general debugging program. Finally, these protocols hide much of the networking details so that other underlying protocols (*e.g.*, ISO/OSI) may be used in place of TCP/IP.

Chapter 3

Application Services of MMWB

3.1. Introduction

MMWB implements an integrated molecular modeling system on a heterogeneous network. The many services provided by **MMWB** are distributed among the hosts in the network. Many hosts may provide the same service, *e.g.*, data management. A single host may provide several services, *e.g.*, data management and energy minimization. Servers (specific instances of services) may call upon each other to accomplish a set of tasks, *e.g.*, an energy minimizer requesting data from a data manager. A group of servers act cooperatively to interact with the user.

The services provided by **MMWB** may be divided into three major categories: those that supply data, those that transform data, and those that present data. In more familiar terms, the categories are data management, computation, and user interface. Different types of services have very different operating characteristics. The common thread among all these services is the inter-service communications described in

Chapter 2.

3.2. Data Management

Data management is at the heart of a network-based system. Without a well-defined and well-controlled data access method, the system may easily fragment into several isolated subsystems with no means of communications. The important issues in data management are data organization, data access protocol, concurrency control, and crash recovery.

MMWB ignores one major aspect of data management: selective data retrieval. This design choice was selected because molecular databases can fit in main memory of modern computers, and memory-based algorithms may be used for data management. Without the limitation of long disk access times, memory-based data management is much more efficient than disk-based database management. Thus, **MMWB** data managers provides facilities to load entire data sets into main memory but leave selective data retrieval to individual applications.

3.2.1. Data Organization

Data used by **MMWB** must be organized in such a way that they can be exchanged among applications residing on different hosts. One possibility is to make data organization closely parallel a data construct in a particular programming language, *e.g.*, array of records in Pascal. However, this choice is certainly sub-optimal for some environments, *e.g.*, Fortran programs prefer parallel arrays. A higher level data abstraction which is independent of programming environment is preferred.

One possible way to organize data is to use tables, an approach previously used by the Prophet System for managing small molecules and associated data [8]. A table consists of a series of rows, each describing a different instance of an item. Each row consists of a number of columns, each describing a different attribute of the item. Thus, a table may describe the atoms in a molecule using the coordinate, name, and sequence number attributes. Tables are particularly well suited for molecular data because molecular data frequently consist of descriptions for many instances of the same type of item, *e.g.*, atoms, residues, connectivity.

The concept of tables is not tied to any particular programming language construct. Tables may be implemented using parallel arrays in Fortran, array of structures in C, or list of lists in Lisp. This flexibility is important in the heterogeneous network environment in which **MMWB** operate.

3.2.2. Data Access Protocol

Defining the data access protocol is straightforward once the data abstraction has been selected. Each command in the access protocol corresponds to an operation on the data abstraction. The data for commands or replies to commands consists of items from the data abstraction.

For tables, the defined operations are:

- get list of tables
- get table description
- get column description
- get column data
- get row data
- insert row
- insert column
- delete row
- delete column
- update row
- update column
- create table
- destroy table

and the items within table are:

- table identifiers
- column identifiers
- table description (number of rows and columns, and column identifiers)
- column description (data type and size)
- data stored in tables
- list of table identifiers (stored in a table, of course)

The detailed data access protocol is described in

Chapter 4.

3.2.3. Concurrency Control

In university research environments, where much of molecular modeling is done, concurrency control is not a critical issue.[4] In general, users tend to work on distinct and non-intersecting projects and data sharing is rare. In view of this situation, **MMWB** can implement concurrency control by enforcing the simplistic policy of permitting at most one data manager to have write-access to a data set at any given time.

This approach of single write-access to data sets has its merits and shortcomings. A clear disadvantage is that orthogonal services (*i.e.*, services which will not modify any data other services access) which potentially can execute in parallel must be serialized. For example, a user may want to perform two different computations on the same data set and store both sets of computed data back into the data set. A less obvious weakness of this approach involves whether other data managers can gain read access to the data set while one holds write access. If other data managers can read the data set, they may potentially read the data set while it is being updated and get an incorrect version of the data set. If, on the other hand, other data managers are not permitted to access the data set, then much concurrency is lost. All these problems may be solved by requiring processes that access the same data to be in the same **conversation**. Then, all the processes can share the same data manager and the multiple access problems disappear.

The advantages of the single write-access scheme derive from its simplicity. Because it is simple, it is easy to implement and manage. Additionally, since write access to a data set may be granted or rejected at file opening time, data managers have

---

[4] In industrial environments, where researchers work in large teams rather than individually, the concurrency problem must be addressed in a more sophisticated manner. The solution proposed at the end of this section is a first step towards solving the multi-user concurrent access problem.

very little concurrency control overhead. Services which invoke data managers are also freed of the responsibility to constantly keep data sets in consistent states. These advantages outweigh the disadvantages because the disadvantages only apply in limited circumstances while the advantages apply in all cases.

3.2.4. Crash Recovery

Given the simple concurrency control scheme described above, crash recovery is a relatively simple task. Most data updates are grouped into transactions, where each transaction modifies a data set from one consistent state to another. Since the **MMWB** concurrency control scheme only permits one data manager to access a data set at any given time, one can treat the entire access as one single monolithic transaction. Thus, updates are sequential non-overlapping operations on the data set. To guarantee data set integrity, all one needs to do is require that data managers not overwrite the original data set when updating it. Instead, a second copy is written, verified, and renamed as the new version of the data set. If a crash occurs during the write or verification process, the original data set is still available. If a crash occurs during the renaming process, then either the original or new version of the data set may be recovered.

3.3. Computation

Computation consists of many different types of applications. They range from geometric calculations such as computing the solvent-accessible molecular surface to energy calculations such as molecular mechanics. The data required for these computation range from atom type to partial charges of atoms. The output range from a set of coordinates for each atom to modified coordinates for each atom. Their widely varying

operating characteristics make standardizing computation applications only possible at a very general level.

Computation servers in MMWB can generally be modeled with the following sequence of actions:

1) process user-specified options
2) fetch molecular data
3) process molecular data
4) store computed quantities

With this limited model, only a few simple guidelines may be specified for computation servers acting as servers in MMWB. User-specified options are normally either boolean flags, *e.g.*, whether to use all atoms of a molecule in a surface calculation, or parameter values, *e.g.*, the density of points in the surface calculation. In either case, the amount of data needed to specify the options is very limited. Thus, these options should be implemented as commands in the communications protocol between client and server. Molecular data, on the other hand, are generally quite voluminous. Although passing such data as part of commands is possible, a better solution is to pass the molecular data via a data manager. Passing data via data managers has two distinct advantages over passing data directly: first, both client and server may take advantage of the operations provided by the data manager, and second, the communications protocol is simplified because no new data format needs to be defined. Output from computation servers vary from a single number, *e.g.*, the total energy of a molecule in a certain conformation, to a great many numbers, *e.g.*, the atomic coordinates of the minimum energy conformation. Generally, low volume output should be sent as replies to commands while high volume output should be sent to a data manager.

3.4. User Interface

The user interface is the part of the system which interacts with the user. As such, it greatly influences the chance of success of the system. While a good user interface might be taken for granted, a bad one may spell doom for the system. The user interface is constrained by the input and output devices available to the system. In a network environment, many types of devices are available and the goal in designing user interfaces is consistency. Other important goals are ease of use, efficiency, and extensibility.

Display capabilities of the output device constrains how data may be presented to the user. Alphanumeric terminals provide very limited methods of displaying data, using only alphanumeric characters and punctuation, while graphics workstations provide very powerful methods, using lines, polygons, and raster images. These constraints greatly influence how well each type of device supports different user interfaces.

User interfaces can generally be divided into two types: graphical and alphanumeric. Graphical interfaces are powerful for displaying large amounts of data in interpreted form, *e.g.*, atomic coordinates as **Corey-Pauling-Koltun** or ball-and-stick models which are easier to interpret than thousands of numbers. Graphical interfaces are excellent at presenting a global view of a set of data but poor at presenting precise information. Alphanumeric interfaces, on the other hand, are good at presenting precise information as strings of characters. However, due to the limited size of terminal displays and human short-term memory, they are poor at presenting global trends.

Input devices place additional constraints on user interfaces. The classic input device is the keyboard, which may be used to either enter commands or data. Keyboards are useful for interacting with alphanumeric displays but are inefficient for interacting with

graphical displays. Pointing devices, such as a mouse or light pen, provide ways for users to indicate areas of interest on the display without knowing names associated with the displayed items. They also provide an alternative to the typing interface of a keyboard: menus. Pointing devices are generally available only on devices capable of some graphics, thus further distinguishing alphanumeric and graphical displays.

Due to the very different characteristics of graphical and alphanumeric devices, two different types of user interfaces are required to make full use of their different capabilities. Since most molecular modeling applications deal with large amounts of atomic data, the primary interface to **MMWB** is its graphics display program, **MIRAGE** (**M**olecular **I**nteractions **R**epresented **A**s **G**raphical **E**ntities). The alphanumeric interface is used to access uninterpreted data residing in data sets or to access **MMWB** services at the communications protocol level.

3.5. Summary

The services provided by **MMWB** can be divided into three general categories with very different operating characteristics. The issues that each category of services must consider are listed in Table 1.

Data Management	Computation	User Interface
Data organization	Data access protocol	Ease of use
Data access protocol	Service-specific considerations	Efficiency
Concurrency control		Extensibility
Crash recovery		Consistency

Table 1 – Service categories and design issues

Data management in **MMWB** depends on the fact that molecular data can fit in the main memory of modern computers. This critical assumption greatly simplifies data

management since data managers no longer need to be concerned with selective data retrieval. Instead, molecular data may be treated as several large tables whose contents need not be examined by the data manager. With tables as the abstraction for molecular data, the communications protocol for the data manager reduces to a set of commands corresponding to operations on tables. Another simplifying assumption for data management in **MMWB** is that only one data manager may access a data set at any given time. This assumption makes both concurrency control and crash recovery trivial issues.

Computation services comprise the most diverse category due to the complex nature of molecular modeling. The general guidelines for implementing computation services in **MMWB** are

- Define a precise communications protocol
- Implement options and parameters as commands
- Pass molecular data via data managers
- Return low volume output as replies to commands
- Return high volume output via data managers

These recommendations are intended to create a simple and easily reproducible execution environment for the computation server so that debugging is simplified.

User interface is an important part of the system because it is the only part that users see. **MMWB** supports two different interfaces for graphical and alphanumeric devices. The need for different interfaces is due to the very different characteristics of the two types of devices. The primary interface to **MMWB** is **MIRAGE** because most molecular modeling applications deal with voluminous atomic data.

Three classes of services, data management, computation, and user interface, form the **MMWB** molecular modeling system. Acting cooperatively, they provide easy and complete access to resources available in a heterogeneous network environment.

Chapter 4

Data Management in MMWB

4.1. Introduction

Techniques from many fields, such as energy minimization and distance geometry, are used to model molecules. These techniques share some data, such as atomic coordinates. Each, however, requires its own exclusive data as well, *e.g.*, only energy minimization needs to know the partial charge and non-bonding energy parameters of all types of atoms. The responsibility of the data management system is to provide all these applications with necessary data in a usable form without burdening them with irrelevant data.

Two obstacles that MMWB data management must overcome arise from the heterogeneous network environment. In such an environment, data sets and applications potentially reside on different hosts. The first obstacle is for applications to retrieve data across the network. The second obstacle is for applications to interpret the data once they have been retrieved. Since hosts are not required to be binary compatible, the interpretation process may be quite difficult, *e.g.*, translating IEEE floating point format to VAX floating point format.

MMWB addresses these data management problems by creating a model of molecular data. The model includes both tables and permissible operations upon these tables. The **MMWB data manager** (DM) manages these tables. The operations are the basis for defining a data access protocol to the DM in the format described in

Chapter 2. To provide higher level structure to the data, tables are further organized into an **object file system** (OFS). Application programs then use a functional interface to the

OFS to access molecular data. The following sections describe the data model, **DM**, and **OFS** as well as some implementation issues.

4.2. Data Model

Most molecular modeling data may be described by the entity-relationship diagram shown in Figure 1.

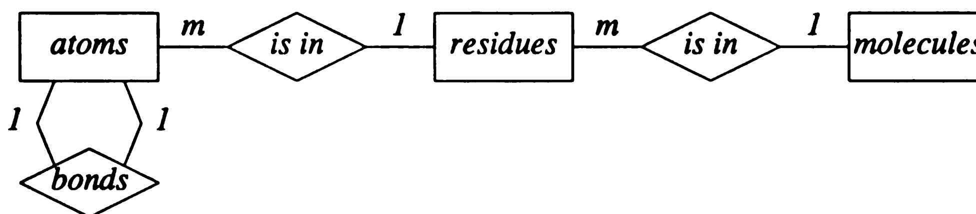


Figure 1 – Entity-relationship diagram for molecular data.

where the rectangles represent entities and diamonds represent relationships. The connecting line segments are labeled with either '*1*' or '*m*' to show whether relationships between entities are one-to-one, many-to-one, or many-to-many. Both entities and relationships may have associated descriptive attributes, such as coordinates for atoms and hydrophobicity for residues. The attributes are not shown in the diagram above to preserve readability.

The organization described by the entity-relationship diagram provides a method for managing the molecular data in a uniform manner for many different modeling applications. The molecular data required by each application are a subset of the attributes of entities and relationships. If attributes may be selectively retrieved, modified, added, and deleted, all applications can conveniently share the same set of data.

The scheme described by Figure 1 may be implemented using tables. Each table corresponds to an entity or relationship. The rows of tables correspond to *instances* of

the entities or relationships; the columns of tables correspond to *attributes*. The intersection of an instance and an attribute is a *cell*, where actual information is stored. New attributes may be added by adding new columns to tables. These tables are similar to relations of relational databases without the restrictions of normalization[5].

Tables may either be ordered (instances are identifiable by row number) or unordered (physical positioning within tables of instances is not significant). Although the latter model gives greater flexibility to the data model, it also imposes certain penalties which make it undesirable. For example, instance operations must identify the target by the value of an attribute. This implies that the data manager needs to know about the concept of key attributes which must be unique within a table. Additionally, accesses to instances, either reading or writing, will also be slower since the data manager must now search the table based on the value of the key attribute. These characteristics make unordered tables less attractive than ordered tables.

Instances in ordered tables are identified by row numbers. This implies that the data manager does **not** need to know the value of attributes in the table. All accesses can be done without data references. The simplicity of the data model makes it much more attractive than unordered tables. The major disadvantage of ordered tables is that instances are **pinned** in ordered tables, *i.e.*, once an instance is known to be in row *n*, it must remain at row *n* until there are no further references to it. Pinned instances make compacting sparse tables difficult (though not impossible), and forces a different interpretation on row deletion. In unordered tables, a deleted instance simply disappears; in ordered tables, the deleted row is still present, but references to its attribute values

[5] Although the data manager does not enforce normalization restrictions, applications are encouraged to use a normalized data scheme.

result in errors.

4.2.1. Items in the Data Model

A complete description of a table consists of

- a table identifier
- the number of instances stored in the table
- the number of attributes
- descriptions of attributes
- the actual entries in the table

The table identifier is used to distinguish one table from another within the same data set. A table identifier together with a data set name uniquely identifies a table on a host. The number of instances and attributes describe the size of the table. The attribute descriptions specify the format of entries in the table. The actual entries in the table comprise the actual data.

The description of attributes consists of

- attribute identifier
- attribute type

The attribute identifier, used to distinguish one attribute from another within the same table, is a string of alphanumeric characters. The attribute type determines how the attribute is described. Attribute types are divided into three categories: **simple, composite,** and **reference**. Simple types include:

- 1-, 2-, 4-, and 8-byte integers
- 4- and 8-byte floating point numbers
- 1-byte characters

Composite types consist of fixed-length arrays with elements of simple type and variable-length strings of characters, which are arrays of characters terminated by a null character (ASCII 0). Reference types are used to refer to other items in the data set. The

only reference type that the data manager supports is the **table reference** type. Although the reference type may be implemented in many different ways internally, the external representation is the same as table identifiers.

Tables, instances, and attributes are the items in this data model. The following section discusses the operations, which define the set of permissible actions upon these items.

4.2.2. Operations in the Data Model

Operations, which are actions performed upon items, may generally be divided into four types. The first type is synchronization operations, *e.g.*, locking. The second type is data description operations, *e.g.*, enumerating existing tables. The third type is data manipulation operations, *e.g.*, creating tables or reading data. The last type is data set operations, *e.g.*, creating or deleting data sets.

4.2.2.1. Synchronization Operations

The first goal of synchronization is to prevent several data managers from modifying the same data set simultaneously. Although it is possible to coordinate updates among several data managers, the required synchronization operations would necessarily be complex. Since data sharing among unrelated tasks is deemed a rare event, the data managers are required to have exclusive access to a data set before operating on it. Exclusive access may be achieved by designating a specific file to as the **access file** for a data set and using file locking. Only the data manager which successfully locks the access file may use it; all others must either give up or wait until the file is relinquished, except in one case. The exception occurs when the data manager which failed to lock the

access file is in the same conversation as the one which succeeded. This may happen if the file has multiple names (*e.g.*, multiple links to the same file, or an entire file system is accessible from several hosts via network file system services) and two data managers tried to lock the same file using different names. In this case, it is desirable that the failed data manager notify its client about the location of the active data manager.

The second goal of synchronization is to enable orderly access to data among clients of a single data manager. The problems caused by uncoordinated data access apply equally well to clients as to data managers. Unfortunately, while data sharing among data managers is rare, table- and attribute-sharing among clients is quite common. Thus, the exclusive access solution used for data manager is inadequate when applied to clients. Instead a more elaborate scheme must be used. There are three situations that must be considered:

- Two clients from different conversations try to access the same table or attribute.
- Two non-communicating clients from the same conversation try to access the same table or attribute.
- Two communicating clients from the same conversation try to access the same table or attribute.

The first case cannot occur because data managers only serve clients from the same conversation. Clients from other conversations are rejected before they can attempt to access data. The second case may be handled by standard locking techniques. The most common technique is to permit multiple clients to hold read locks on the same item simultaneously, so they can all read data. If a client wants to modify data, it must acquire a write lock on the item, at which time no other client may hold any lock on the item. The third case is the most difficult and is best illustrated by Figure 2.

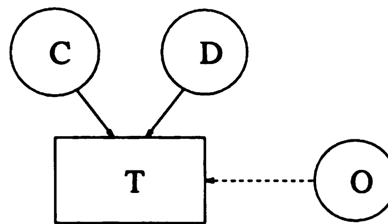


Figure 2 – Clients C and D are cooperatively accessing table T
and do not want client O to interfere.

Suppose two clients are a display program, *D*, and a multi-stage computation program, *C*. *D* wants to show all intermediate results of *C*, but *C* does not want to risk losing its write lock (to *O*, yet another computation program) by relinquishing it each time *D* needs to look at the data. A reasonable solution is to allow both *D* and *C* to hold write locks to the data and let the two clients negotiate between themselves about synchronization. Although this solution appears to be sidestepping the issue, it, in fact, is the most reasonable solution. Only the clients themselves know the level and type of synchronization that is required. Instead of imposing a possibly inadequate locking scheme, the data manager lets the application-level clients control the synchronization method. The problem with this solution is determining which set of clients may simultaneously hold write locks to the same data.

The simplest way to determine whether clients can simultaneously hold write locks to the same data is by associating a key with each lock. The first time a data item is locked for writing, the data manager creates a key for the lock and gives it to the client requesting the lock. This client may then pass the key to other clients. Further write lock requests to the data manager for this data item will succeed if the proper key is presented, and fail if an improper key or no key is presented. As with synchronization, key distribution policy is left to application-level clients, thus providing great flexibility to the

clients. This setup is similar to the capability schemes used by some operating systems [5].

In summary, data managers provide synchronization operations at two levels: among data managers and among clients. Among data managers, synchronization is achieved by exclusive file locking. Although primitive, exclusive access is sufficient because data sharing among data managers is uncommon. Among clients, simple synchronization is achieved by a hybrid of locking and capability schemes. More complex synchronization is left to the discretion of application-level clients.

4.2.2.2. Data Description Operations

The **DM** provides two data description operators: **size** and **enumerate**. The **size** operator lists the dimensions of an item. The **enumerate** operator lists the description of an item.

The **size** operator, when applied to a data set, returns the number of tables in the data set. When applied to a table, it returns two values: the total number of attributes, and the number of attributes with at least one cell of data. When applied to an attribute, the **size** operator also returns two values: the number of cells with data, and the maximum row number of cells with data. The former dimension describes how big a table is; the latter describes how much of it is used.

The **enumerate** operator parallels the **size** operator. When applied to a data set, it returns a list of table names. When applied to a table, it returns a list of attribute name and their associated types. When applied to a attribute, the **enumerate** operator returns a list of row numbers of cells with data.

4.2.2.3. Data Manipulation Operations

There are four data manipulation operations: create, delete, get, and put. Each operation may be applied to any item. The semantics of each operation is straightforward and, hopefully, intuitive. Table 2 describes the information needed for each operation and required conditions for success.

Operation	Required Information	Condition for Success
Create Table	Table identifier	Table identifier is not already in use.
Create Instance	Table identifier	Table is locked for writing.
Create Attribute	Table identifier Attribute identifier Attribute type	Table is locked for writing and attribute identifier is not already in use.
Delete Table	Table identifier	Table is locked and table identifier exists.
Delete Instance	Table identifier Row number	Table is locked for writing and all attributes are locked for writing.
Delete Attribute	Table identifier Attribute identifier	Table is locked for writing and attribute exists.
Get Table	Table identifier	Table is locked.
Get Instance	Table identifier Attribute identifiers Row number	All attributes are locked.
Get Attribute	Table identifier Attribute identifier	Attribute is locked.
Put Table	None	The **put** operation is not permitted on tables.
Put Instance	Table identifier Attribute identifiers Row number Attribute values	Attributes are locked for writing.
Put Attribute	Table identifier Attribute identifier Attribute values	Attribute is locked for writing.

Table 2 – Operations in the Data Model

4.2.2.4. Data Set Operations

The operations listed in the previous section dealt with data within data sets. Operations which manipulate data sets as complete entities are also needed for data management. The four operations supported by the data manager are **open**, **close**, **create**, and

delete data sets.

4.3. Data Organization

Although tables are suitable for storing molecular data, they have two shortcomings. First, the only way to determine the semantic meaning of the contents of a table is to examine the attribute descriptions. Second, while tables neatly map to many aspects of molecular data, they do not provide data organization for manipulating molecules as complete entities. Both these problems may be easily remedied. The first problem may be solved by adding a new field to the table header called **class**. The class of a table determines the minimal set of attributes that must be present in the table. The second problem may be solved within the current framework by creating new tables that contain references to all existing tables relevant to a particular molecule.

These two solutions go part way toward a much more general concept. Tables store the raw molecular data. Collections of tables can be organized into **objects**. This multilevel organization is similar to handling written correspondence. Tables are analogous to letters, and objects to files of letters. All letters exchanged with a particular correspondent may be dealt with using a single file. The advantage of using the object concept is that objects will match physical items, such as molecules. Thus, application libraries may take advantage of the organization to implement the object-specific semantics. The desired end result is that users will manipulate familiar items rather than fragmentary data.

An object can be described completely by a set of properties. This structure maps directly to a single-instance table, with the attributes of the table corresponding to the properties of the object. Operations on the table are also operations on the object. The

entire object abstraction is designated the **object file system, OFS.**

4.3.1. Object Properties

The minimal set of properties of objects is shown in Table 3.

Property	Description
Type	The type of this object (*e.g., protein*).
Class	The class of this object (*e.g., molecule*).
Name	The name that a user specifies for this object.
ACL	The Access Control List that controls which users have access to this object.
Owner	The name of the user who controls the ACL of this object.
Time	Creation, modification, and access times.
Description	Description of what this object really is.
Notes	Additional descriptions.
Position	Screen location of this object in some normalized coordinate system.
Parent	The object that includes this object.
Data	Raw (numeric) data associated with this object.

Table 3 – Typical properties of objects.

Objects usually has more properties than those listed in Table 3. The additional properties depend on the value of the **type** property. For example, an object of type **molecule,** as described by Figure 1, will also have properties **residues, atoms,** and **bonds.**

4.3.2. Object Operations

Since an object is described by a table, operations which apply to table attributes, *e.g.*, add, delete, get, put, *etc.*, are also applicable to object properties. However, objects have more operations than tables because a minimal set of properties are known to exist for object. Table 4 lists the object operations.

Function	Description
Open pathname	Fetch objects stored in *pathname*.
Enumerate objects	List all known objects.
Add object	Create a new object.
Delete object	Destroy an existing object.
Find object	Locate an object by property values (*e.g.*, name).
Set default object	Make property list operations refer to this object.
Drop object	Change the default object to nil.
Enumerate property list	List properties of default object.
Get property	Get the value of a property.
Update property	Change the value of a property.
Add property	Create a new property.
Delete property	Destroy an existing property.

Table 4 – Object operations

4.3.3. Projects

A collection of objects may be grouped further into more complex **aggregates**. For example, a set of related objects may be grouped into a **project**, which is a special type of aggregate. Users may then deal with data on a project-by-project basis, perhaps in a hierarchical structure, rather than trying to handle a huge flat data space.

4.4. Interface Specifications

MMWB data management specifications consists of two parts: program functional interface and network protocol interface. The program functional interface describes the list of procedures that applications use to access data. The network protocol interface describes the protocol that the functional interface uses to communicate with data managers across the network.

4.4.1. Program Functional Interface

The functional interface defines several new data types and a plethora of pro-

cedures. The interface is specified in the C language. Bindings for other languages such as Fortran are will be formulated at a later date.

4.4.1.1. Data Types

Some of the new data types are opaque, *i.e.*, applications need not be concerned with how the type is implemented. Other data types are transparent, *i.e.*, types whose structure must be known by applications. Opaque types are generically described as type `void *`. The C declarations for these types are listed below.

```
/*
 * Opaque types
 */
typedef void    *DM_HANDLE;     /* Data manager handle */
typedef void    *DM_TABLE;      /* Table handle */
typedef void    *DM_ATTR;       /* Attribute */
typedef void    *DM_OBJECT;     /* Object handle */
typedef void    *DM_PROP;       /* Property */
typedef int     DM_KEY;         /* Key to locks */

/*
 * Transparent types
 */
typedef int     DM_ATYPE;       /* Attribute type */
#define DM_A_EXIST      0       /* ..Any attribute */
#define DM_A_FILLED     1       /* ..Attribute with any data */
#define DM_A_FULL       2       /* ..Attribute with all data */

typedef int     DM_ITYPE;       /* Instance type */
#define DM_I_MAX        0       /* ..Maximum instance number */
#define DM_I_FILLED     1       /* ..Number of filled instances */

typedef int     DM_LMODE;       /* Lock mode */
#define DM_READLOCK     0       /* ..Lock for reading (shared) */
#define DM_WRITELOCK    1       /* ..Lock for writing (exclusive) */

typedef struct {                /* Storage type */
        int     base_type;      /* ..Basic element type */
        int     length;         /* ..Elements per datum */
}               DM_STYPE;
#define DM_S_INT1       0
#define DM_S_INT2       1
#define DM_S_INT4       2
#define DM_S_INT8       3
#define DM_S_FLOAT4     4
#define DM_S_FLOAT8     5
#define DM_S_FLOAT16    6
#define DM_S_CHAR       7
#define DM_S_STRING     8
#define DM_S_BYTE       9
#define DM_S_TABLE      10

typedef struct {                        /* Object description */
            int         type;                   /* ..Object type */
            int         class;                  /* ..Object class */
            int         ident;                  /* ..Unique identifier */
            char        *name;                  /* ..External object name */
}           DM_OBJ_DESC;

/*
 * Some constants for simplifying implementation
 */
#define     DM_TABLE_NAMELEN        255
#define     DM_ATTR_NAMELEN         255
```

4.4.1.2. Procedures

The procedures are divided into three sections: data manager, table, and object routines. Each routine is listed with its return value and its calling convention. The variables and arguments used by the routines are listed in Table 5.

Variable	Type	Description
acct	char *	Remote account name
ah	DM_ATTR	Attribute handle
attr	char *	Attribute name
atype	DM_ATYPE	Attribute type
cid	char *	Conversation identifier
class	int	Table class
data	void *	Pointer to data buffer
dmh	DM_HANDLE	Data manager handle
fp	FILE *	Pointer to stdio stream
host	char *	Name of host where data set resides
itype	DM_ITYPE	Instance type
key	DM_KEY	Key to locks
lock_mode	DM_LMODE	Lock mode
n	int	Generic integer
name	char *	Name of table, object, or attribute
names	char **	List of pointers to names
oclass	int	Object class
oh	DM_OBJECT	Object handle
oident	int	Object identifier
olist	DM_OBJ_DESC	Object description list
otype	int	Object type
path	char *	Pathname on host where data set resides
prop	char *	Property name
s	char *	Arbitrary character string
sf	int	Return status, negative=failure, zero=success
stype	DM_STYPE	Storage data type
th	DM_TABLE	Table handle
wait	int	Whether lock functions should wait

Table 5 – Functional specification variables and types

The return value of each routine indicates the execution status of the function call. Null pointers and negative numbers indicate failure; valid pointers and zero indicate success.

A global variable contains more detailed description of the error. The possible error values are listed below.

```
int      dm_errno;
#define DM_ENOERROR     0     /* No error */
#define DM_ESYSERROR    1     /* See <sys/errno.h> for error */
#define DM_ETOOSMALL    2     /* User-supplied buffer is too small */
#define DM_ENOPERM      3     /* Permission denied */
#define DM_NODATA       4     /* Accessed cell was empty */
#define DM_CANTLOCK     5     /* Cannot immediately lock item */
```

4.4.1.2.1. Data Manager Routines

The following sections detail the various functions and procedures used to communicate with the data manager.

```
dmh = dm_open_path(cid, acct, host, path);
 sf = dm_close_path(dmh);
dmh = dm_create_path(cid, acct, host, path);
 sf = dm_delete_path(cid, acct, host, path);
 sf = dm_times(dmh, atime, mtime);
```

These functions are used to obtain and release access to data sets. It is an error to open a non-existent data set; it is also an error to create a data set when one already exists. The last function is used to get the last access and modification time of the data set.

```
name = dm_path_id(dmh);
 dmh = dm_path_join(name);
```

The first function is used to obtain a data manager identifier which may be used as the argument to **dm_path_join** for getting a second connection to the same data manager. These functions are most useful when one DM client starts up another client and needs to tell the latter where to get shared data.

```
(void) dm_perror(fp, s)
(void) dm_sync(dmh);
```

dm_perror is used to generate error messages based on the current value of **dm_errno**.

dm_sync is used to ask the functional interface to flush any locally cached data to the remote data manager process.

4.4.1.2.2. Table Routines

```
n = tbl_count(dmh);
n = tbl_count_by_class(dmh, class);
n = tbl_list(dmh, names, n);
n = tbl_list_by_class(dmh, class, names, n);
```

These functions are used to obtain a description of the data set. Note that the **list** functions fill in data areas provided by the caller. If the data area is too small, the functions fill in the entire data area and return an error; otherwise, they return the actual number of names filled in.

```
   th = tbl_open(dmh, name);
   sf = tbl_close(th);
   th = tbl_create(dmh, name, class);
   sf = tbl_delete(th);
class = tbl_class(th);
 name = tbl_name(th);
```

These functions are used to gain and release access to tables, as well as getting the table class and name. As with data sets, it is an error to open non-existent tables or attempt to create an already existing table.

```
key = tbl_lock(th, lock_mode, key, wait);
 sf = tbl_lock_change(th, lock_mode, key, wait);
 sf = tbl_unlock(th, key);
```

These functions are used to explicitly manage the locks on tables. Locks are ''stackable''. If a table is locked twice by the same client, it must be unlocked twice to be released. A **key** value of **NULL** means that the caller currently holds no key. The caller can either wait for lock operations to complete or get an immediate failure if the lock is currently unavailable by setting the **wait** to either non-zero or zero respectively.

```
n = attr_count(th, atype);
n = attr_list(th, atype, names, n);
```

These functions are used to obtain the description of a table. The **atype** argument is used to determine which attributes of the selected table will actually be counted or listed. The value is interpreted as follows:

DM_A_EXIST	All attributes are used
DM_A_FILLED	Only attributes with associated data are used
DM_A_FULL	Only attributes with data associated with every single cell are used

As with the table functions, the list function fills in the data area as much as possible; if the area is too small, it returns an error.

```
  sf = attr_create(th, attr, stype);
  ah = attr_open(th, attr);
  sf = attr_close(ah);
  sf = attr_delete(ah);
name = attr_name(ah);
```

These functions are used to manipulate attributes. A storage type must be supplied when an attribute is created. If instances are already present in the table, they are extended to include the new attributes and the new cells are empty. Both **attr_create** and **attr_open** automatically acquire a read lock on the specified attribute. Both **attr_close** and **attr_delete** release any lock held for the specified attribute. **attr_name** returns the name of the attribute.

```
key = attr_lock(ah, lock_mode, key, wait);
 sf = attr_lock_change(ah, lock_mode, key, wait);
 sf = attr_unlock(ah, key);
```

These functions are used for explicit management of locks on attributes. As with table locks, these locks are also stackable.

```
stype = attr_stype(ah);
    n = cell_count(ah, itype);
```

These functions are used to obtain the description of an attribute. The storage type of the attribute may be used to allocate memory for data areas in calls to **cell_read** and **cell_write**. If an error occurs, the base type of **stype** will be set to -1. The second function is used to determine the number of cells under the attribute. If the **itype** argument is **DM_I_MAX**, then the greatest instance number is returned; if **itype** is **DM_I_FILLED**, then the total number of filled cells is returned. These functions implicitly lock the table and attribute before executing and release the locks when done.

```
 n = inst_create(th);
sf = inst_delete(th, n);
```

These functions are used to create and destroy instances. A newly created instance always has an instance number greater than any existing instance. The cells in the new instance are all empty. When an instance is deleted, all cells in the instance are marked as empty. No other action is taken unless the deleted instance had the greatest instance number in the table, in which case its storage space is recovered. These functions implicitly lock the table before executing and release the lock when done.

```
sf = cell_read(ah, n, data);
sf = cell_write(ah, n, data);
```

These functions are the ones that actually access and modify data within tables. These functions implicitly lock the table and attribute before executing and release the locks when done. Note that reading an empty cell results in an error. **data** is a pointer to the area where data is fetched or stored. For tables, **data** should be the address of a table handle.

4.4.1.2.3. Object Routines

```
n = obj_count(oh);
n = obj_count_by_type(oh, otype);
n = obj_count_by_class(oh, oclass);
n = obj_list(oh, olist, n);
n = obj_list_by_type(oh, otype, olist, n);
n = obj_list_by_class(oh, oclass, olist, n);
```

These routines are analogous to the table counting and listing functions. The only difference is that the **otype** and **oclass** arguments refers to the object type and class rather than table class.

```
    oh = obj_open_by_name(oh, name);
    oh = obj_open_by_type(oh, otype);
    oh = obj_open_by_class(oh, oclass);
    oh = obj_open_by_ident(oh, oident);
    sf = obj_close(oh);
    oh = obj_create(oh, name, otype);
    sf = obj_delete(oh);
 otype = obj_type(oh);
oclass = obj_class(oh);
  name = obj_name(oh);
```

These routines are analogous to the table access functions. The only difference is that objects may be opened either by **object** name or type. Note that these are not the name or class of **tables**.

```
key = obj_lock(oh, lock_mode, key);
 sf = obj_lock_change(oh, lock_mode, key);
 sf = obj_unlock(oh, key);
```

These functions are used for explicit management of locks on objects. They are usually unnecessary as most object handling functions implicitly lock objects automatically. They are provided for completeness.

```
n = prop_count(oh);
n = prop_list(oh, names, n);
```

These functions are analogous to the table description functions.

```
stype = prop_stype(oh, prop);
```

This function returns the storage type of a property. Note that there is no function analogous to **attr_count** as properties only have a single value associated with them.

```
sf = prop_create(oh, prop, stype);
sf = prop_delete(oh, prop);
sf = prop_read(oh, prop, data);
sf = prop_write(oh, prop, data);
```

These functions are used to access properties. The create and delete functions are analogous to their table counterparts. The read and write functions are analogous to their cell counterparts except no instance number is required.

```
oh = proj_open(oh);
oh = proj_create(oh, ename, host, path);
sf = proj_delete(oh);
sf = proj_close(oh);
```

These functions are used to manipulate projects, *i.e.*, the objects which are used to organize other objects. Each project is associated with a pathname on a particular host. The caller can supply the host name and the pathname, or use the system default by setting **host** and **path** to NULL. The results of these calls are object handles which may be manipulated in the same way as normal object handles.

```
  oh = ofs_sys_init();
  oh = ofs_proj_init(name);
name = proj_id(oh);
```

These are the initialization routines for the OFS. The first function opens a system project where prototype objects and other miscellaneous information are kept. The second function is used to open a user project. The argument to the second function may be either NULL, where the OFS will open an 'appropriate' project, or a value returned by

`proj_id`, where the OFS will open that project.

4.4.2. Network Protocol Interface

The network protocol interface defines the communications protocol used between the functional interface library and the remote data manager process. The protocol consists of commands and responses as described in

Chapter 2. Only responses specific to the data manager are listed below. Generic responses such as "permission denied" and "internal server error" may also be generated.

MACHINE machine_type

This command is the first command issued after the low level verifications (*i.e.*, the **VERS, HELO,** and **USER** commands) have been sent, and is considered as part of the generic handshaking between the client and the data manager. *machine_type* is a quoted string identifying the machine type of the client. If the server recognizes the machine type and can send binary data to that machine type, then bulk data will be sent in binary format (see the **READ** and **WRITE** commands below). The possible responses are

210 Use **ASCII** data
211 Use binary data

OPEN DATASET pathname

OPEN DATASET is used to access a data set. *pathname* is a quoted string that contains the path to the desired data set. The possible responses are

210 *pathname* opened for reading and writing
211 *pathname* opened for reading only
410 *pathname* does not exist

412 *pathname* already opened by another server (This response should be followed by an **ASCII** data response (code 202) containing a string of the form *pathname@host.*)

CREATE DATASET pathname

CREATE DATASET is used to create a data set. This command does **not** implicitly open the named data set. *pathname* is a quoted string that contains the path to the desired data set. The possible responses are

210 *pathname* created for reading and writing
411 *pathname* already exists

DELETE DATASET pathname

DELETE DATASET is used to destroy a data set. The client must have the data set opened in order to delete it. *pathname* is again a quoted string containing the path to the data set. The only possible response is

210 *pathname* is deleted

DESCRIBE DATASET

This command describes the data set. Possible responses are

210 Data set description follows
This response is followed by an **ASCII** data response (code 202). The lines consist of keyword value pairs. The minimal set of keywords supplied is **time,** whose value is two integers: the last modification time and last access time as measured in seconds since January 1, 1970.
410 No such table *table*

COUNT TABLE

COUNT TABLE CLASS class

COUNT TABLE is used to count the the total number of tables. **COUNT TABLE CLASS** is used to count the number of table of class *class*, which is an integer.

The only possible response is

210 Number of tables
followed by
202 ASCII data follow

LIST TABLE

LIST TABLE CLASS class

These commands are used to list table names. The only possible response is

210 List of tables
followed by
202 ASCII data follow

CREATE TABLE table class

This command creates a table with name *table* and class *class*. The possible responses are

210 Table created
411 Table *table* already exists

DELETE TABLE table

This command deletes a table with name *table*. The possible responses are

210 Table deleted
410 No such table *table*

OPEN TABLE table

CLOSE TABLE table

This command informs the data manager whether the client will be accessing *table* in the near future. The **OPEN TABLE** command is also used to verify the existence of *table*. Possible responses are

210 Table opened
210 Table closed
410 No such table *table*
412 Table was not opened

DESCRIBE TABLE table

This command describes a table, listing its class, number of attributes, and number of instances. Possible responses are

210 Table description follows
This response is followed by an ASCII data response (code 202). The lines consist of keyword value pairs. The minimal set of keywords supplied is **class**, whose value is an integer corresponding to the table class; **attributes**, whose value is three integers: the total number of attributes, the number of attributes with associated data, and the number of attributes which are completely filled; **instances**, whose value is an integer corresponding to the row number of the last instance with associated data; **owner**, whose value is the string representing the user identifier of the owner; and **time**, whose value is two integers: the last modification time and last access time as measured in seconds since January 1, 1970.
410 No such table *table*

LOCK TABLE table mode key wait

This command is used to get a *mode* lock to table *table*, where *mode* is either the string **read** or **write**. *key* is the integer key that may have been obtained from a previous lock request. If *key* is zero, then there is no previous key. If the lock is unavailable, the **DM** returns an error message immediately if *wait* is zero;

otherwise, the **DM** waits until the lock is available and sends the success message.

The possible responses are

210 *key* is key to table *table*
410 No such table *table*
413 Cannot lock immediately. Sorry.
414 Key does not match lock
415 Deadlocked

LOCKMODE TABLE table mode key wait

This command is used to change the mode of a currently held lock on table *table*. *mode* and *wait* have the same functions as in the **LOCK TABLE** command. *key* must be a non-zero key from a previous lock request. The possible responses are the same as the **LOCK TABLE** command.

UNLOCK TABLE table key

This command is used to release a currently held lock on table *table*. *key* must be the key from a previous lock request. The possible responses are

210 Table *table* unlocked
410 No such table *table*
414 Key does not match lock

LIST ATTRIBUTE table atype

This command lists the attributes of table *table*. If *atype* is **DM_A_EXIST**, then all attributes are listed. If *atype* is **DM_A_FILLED**, then only attributes with at least one non-empty cell are listed. If *atype* is **DM_A_FULL**, then only attributes with no empty cells are listed. The only possible response is

210 List of attribute names
202 ASCII data follow
410 No such table *table*

CREATE ATTRIBUTE table attribute stype.base stype.length

This command creates an attribute with name *attribute* within table *table*. The storage type for the attribute will be of base type *stype.base* and length *stype.length*. The possible responses are

210 Attribute created
410 No such table *table*
411 Attribute *attribute* already exists
416 Unsupported data type

DELETE ATTRIBUTE table attribute

This command deletes attribute *attribute* from table *table*. The possible responses are

210 Attribute deleted
410 No such table *table*
410 No such attribute *attribute*

DESCRIBE ATTRIBUTE table attribute

This command describes an attribute in the specified table, listing its storage type, maximum instance number, and number of cells with data. Possible responses are

210 Attribute description follows
This response is followed by an ASCII data response (code 202). The lines are keyword-value pairs as in the **DESCRIBE TABLE** command. The minimal supplied keywords are **type**, whose value is two integer corresponding to the *base_type* and *length* fields of the **DM_STYPE** structure; **instances**, whose value is two integers: the total number of cells, and the number of cells with data; **owner**, whose value is two integers: the user and group identifiers of the owner; and **time**, whose value is two integers: the last modification time and last access time as measured in seconds since January 1, 1970.
410 No such table *table*
410 No such attribute *attribute* in table *table*

LOCK ATTRIBUTE table attribute mode key wait

This command is used to get a *mode* lock to attribute *attribute* of table *table*, where *mode* is either the string **read** or **write**. *key* is the integer key that may have obtained from a previous lock request. If *key* is zero, then there is no previous key. *wait* determines whether the **DM** returns an error message immediately if the lock is unavailable, or whether it waits until the lock becomes available. The possible responses are

210 *key* is key to table *table* attribute *attribute*
410 No such table *table*
410 No such attribute *attribute*
413 Cannot lock immediately. Sorry.
414 Key does not match lock
416 Deadlocked

LOCKMODE ATTRIBUTE table attribute mode key wait

This command is used to change the mode of a currently held lock on attribute *attribute* of table *table*. *mode* and *wait* have the same functions as in the **LOCK ATTRIBUTE** command. *key* must be a non-zero key from a previous lock request. The possible responses are the same as the **LOCK ATTRIBUTE** command.

UNLOCK ATTRIBUTE table attribute key

This command is used to release a currently held lock on attribute *attribute* of table *table*. *key* must be the key from a previous lock request. The possible responses are

210 Table *table* unlocked
410 No such table *table*
410 No such attribute *attribute*
414 Key does not match lock

CREATE INSTANCE table

This command creates a new instance in table *table*. The instance number of the new instance is guaranteed to be one greater than the largest instance number previously in use. The possible responses are

210 *n* is new instance number
410 No such table *table*

DELETE INSTANCE table n

This command deletes the contents of all cells in instance *n* in table *table*. If *n* has the largest instance number in use, then a new lower value is found. Note that instance numbers of other instances are unaffected by this command. Possible responses are

210 Instance *n* deleted
410 No such table *table*

READ table attribute from to

This command requests that the **DM** return the value of non-empty cells for attribute *attribute* in table *table* between instances *from* and *to* inclusively. The data may be sent in either ASCII or binary format depending on whether the client and data manager can share binary format data. The return values are split into blocks of consecutive instances. For example, if the client requests data between instances 100 and 200, and there is no data for instance 150, then the **DM** will return the data of instances 100 through 149 first, and then instances 151 through 200. Note that instance 150 is never explicitly mentioned. The possible responses are

210 Values for instances *from to*

202 Data follow
203 *n* bytes of binary data follow
410 No such table *table*
410 No such attribute *attribute*

WRITE table attribute from to

This command requests that the **DM** replace the value of cells for attribute *attribute* in table *table* between instances *from* and *to* inclusively. The actual data should follow the command after the **DM** acknowledges the command. The format of the data depends on the outcome of the **MACHINE** command. The possible responses are

210 Data received
310 Write command okay. Send data.
410 No such table *table*
410 No such attribute *attribute*

EMPTY table attribute from to

This command requests that the **DM** delete the value of cells for attribute *attribute* in table *table* between instances *from* and *to* inclusively. The possible responses are

210 Data deleted
410 No such table *table*
410 No such attribute *attribute*

4.5. Implementation Issues

The previous sections describe the semantics of data manipulation declaratively (what needs to be done), not imperatively (how things should be done). Many different methods may be used to actually implement the data model discussed above. For exam-

ple, a *data set* may map to either a file or a directory in a file system. The techniques that follow are suggestions on how to implement the data model, but are not part of the model.[6]

4.5.1. Implementing the Data Manager

A *data set* should map to a directory in the file system. By using a directory for a *data set*, the implementor can reserve special files within the directory for exclusive use by the data manager. For example, the file *index* may be reserved to store a table which contains the name of all tables in the data set. Other files in the directory can map to tables. Each file should be able to store several tables to limit the number of files to a manageable quantity.

Tables should be stored column-wise for ease of extension. If they were stored row-wise, then addition of columns requires restructuring the file. If they are stored column-wise, then addition only entails appending to the file. In the file, each table is represented by a table header followed by the columns in the table. Each column may then be stored in two different ways. If less than some fraction of the rows in the column has data associated with them, then the column can be stored using sparse matrix techniques, where only existing data are kept [13]. If most of the rows in the column contain data, then a full column may be stored along with a boolean array that marks the empty rows. Tables may be converted from one format to the other with some hysteresis, *e.g.*, tables becoming less than 40% full are converted to the sparse format and tables becoming more than 60% full are converted to the dense format.

---

[6] The source code to a prototype data manager and a functional interface appears in Appendix C.

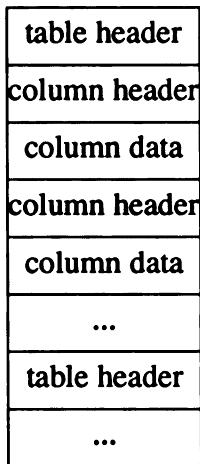


Figure 3 – Internal structure of a file

Data are stored in files as shown in Figure 3. However, when tables are read into memory by the data manager, they are restructured for more convenient access. The sparse matrix columns are converted into hash tables; the mostly full columns are split into segments with equal number of rows. Using hash tables minimizes the access time to specific rows without expending memory for storing nonexistent rows. Using segments makes memory management simple by sacrificing some execution speed. Figure 4 shows the two possible memory arrangements.

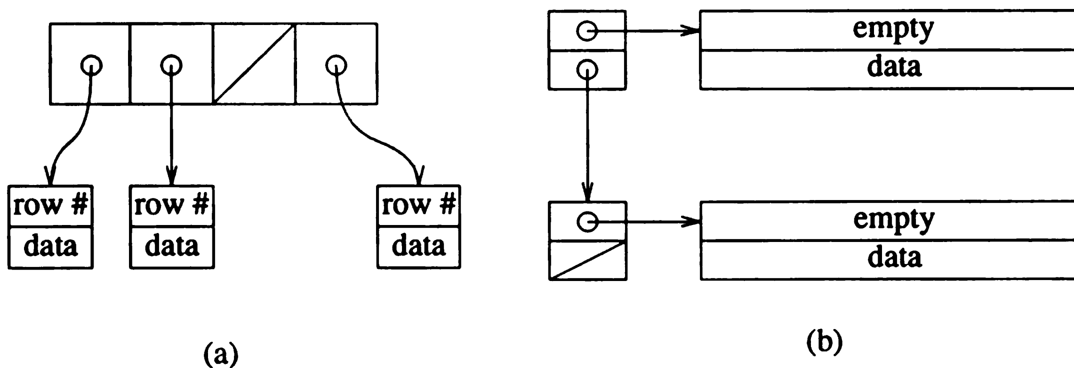


Figure 4 – In-memory data structure of tables:
(a) hash table format for sparse tables,
(b) segmented format for mostly full tables.

The data manager is a multi-threaded server, having many clients sending many requests. If the programming environment provides multiple processes with shared memory and synchronization primitives, the DM may be easily implemented by creating a process for each client and having all processes sharing the actual data. Lock requests are then mapped to synchronization primitives and each individual process may be considered single-threaded.

If the multi-process approach is infeasible due to lack of functionality within the operating system, the data manager may be implemented as a non-preemptive multi-tasking scheduler within a single process. In this approach, the DM is very similar to an operating system, without the hardware interrupts. Clients correspond to processes; tables correspond to shared memory; and locks correspond to synchronization primitives. When a command is received from a client, the data manager executes it until it either completes, and the DM returns the result to the client; or the command blocks waiting for a lock request, and the DM saves the state of the current command, marks it as blocked, and goes on to process other pending commands. When an unlock request is executed, the list of blocked commands are checked to see whether any was waiting for the released lock. If so, those commands are moved back into the list of pending commands. When no commands are pending, the data manager simply waits for further commands from clients. While this approach provides the same functionality as the multi-process approach, it has two shortcomings: first, the implementation is more complex due to the explicit scheduling of command processing; second, there is no possibility for parallel execution of multiple independent commands. These weaknesses make using multiple threads [2, 18] much more attractive.

In either the multi-process or single-process scheme, the data manager must deal with deadlock conditions. While deadlocks are unlikely because separate clients normally access separate tables, the DM should deal with the condition gracefully somehow rather than letting it persist. Since the DM has no control over client behavior, it can only try to detect deadlock conditions and notify the blocked clients. More severe actions such as terminating service to certain clients are also possible.

4.5.2. Implementing the Functional Interface

The data manager only provides the table abstraction. The functional interface must provide the object abstraction to its callers. The similar characteristics of tables and objects make it natural to build one on top of the other. Figure 5 shows how tables may be used to implement objects.

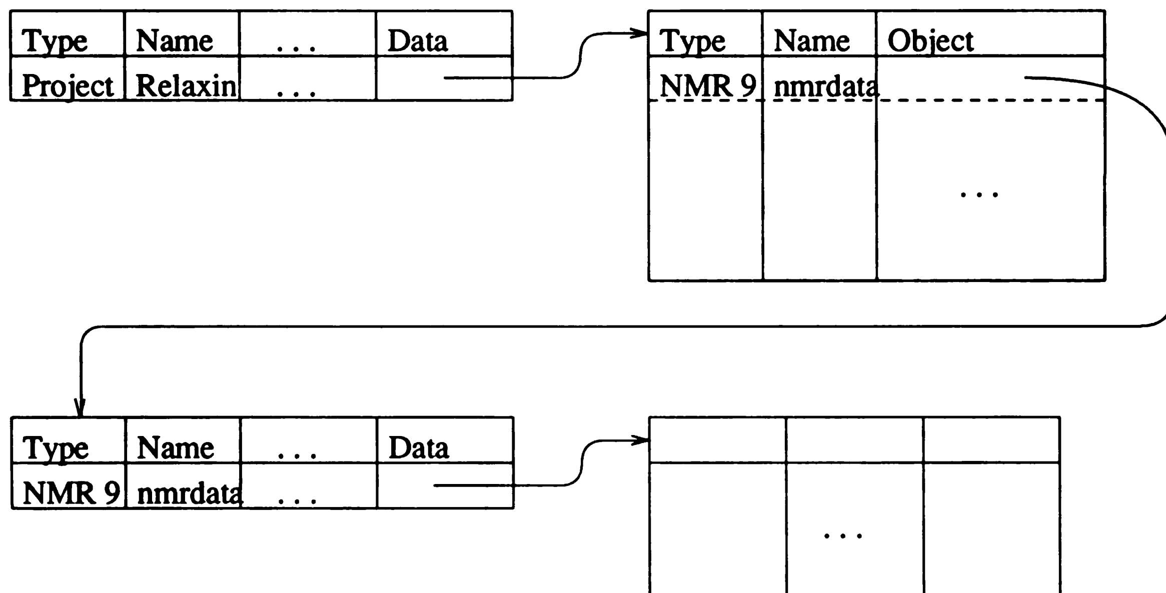


Figure 5 – Objects implemented as tables

In addition to providing the object abstraction, the functional interface can also provide buffered access to the data stored within tables. Instead of always forwarding caller

requests onto the data manager and returning the results, the functional interface can cache data received from the data manager so that further requests for callers may be handled locally. For example, the list of tables may be saved so that description requests from the caller may be answered based on the stored information. By using an intelligent data cache algorithm, the functional interface can minimize network activity and improve overall performance.

4.6. Summary

The data manager employs ordered tables for storing data. Tables are especially appropriate because molecular data translate very easily into tables, and they provide a consistent view of the data across all applications. Tables are easy to understand and manipulate. Also, they are independent of situations, making them equally usable in many programming environments. Finally, the general nature of tables make them an excellent foundation for building higher-level data structures.

The object file system organizes tables into objects that map to physical items. This high level organization facilitates data management by users as well as application programs. As with tables, the object abstraction is also an excellent foundation for constructing higher-level data structures.

Chapter 5

The Programming Framework for MMWB User Interfaces

5.1. Introduction

The primary user interface of **MMWB** is **MIRAGE** (**M**olecular **I**nteractions **R**epresented **A**s **G**raphical **E**ntities). Since the user interface of a system plays a large role in determining the degree of success of the system, the design of **MIRAGE** is an important aspect of **MMWB**.

The overall organization of a program, the **programming framework**, should reflect the goals of the program. The goal of **MIRAGE** is to provide easy access to a variety of **MMWB** services. Thus, the programming framework is evaluated according to extensibility (so new services may be incorporated easily), efficiency (so users need not wait interminably), and development effort and ease of maintenance (so programmers need not work interminably).

There are three possible approaches for implementing **MIRAGE**. The first is an integrated single-layer design. The second is a modular multi-layer design. The third is a hybrid of the first two. Each design is discussed in some detail in the following sections.

5.2. Design Approaches

5.2.1. Single-layer Design

The most straightforward approach to implementing any system is to write the entire system as a monolithic program, where the system commands are rigidly defined by the code. Both the merits and shortcomings of the single-layer approach derives from

system rigidity. The main advantage is that the implementors can select the most efficient algorithm for the system because all system commands are known *a priori*; for example, YACC may be used to generate parsing tables for the command grammar [12]. Additionally, the system is straightforward to maintain because the entire system consists of code.

The main drawback of a single-layer system is its inflexibility. Because system commands cannot be modified at run-time, it must be prepared to deal with any situation that may arise. For a system such as MIRAGE to anticipate all possible applications in molecular modeling is highly unlikely. Another disadvantage to the single-layer system is that programmers are needed to maintain, modify, and enhance the system. Since MMWB will be used at sites which are more concerned with science than computer programming, this shortcoming may have significant repercussions (*e.g.*, many phone calls to the implementors).

5.2.2. Multi-layer Design

A more sophisticated design uses a multi-layer approach. The lowest layer implements system primitives and a *meta*-language; the middle layer implements system commands in terms of system primitives using the *meta*-language; the highest layer interprets user actions and translates them into system commands.

The advantages of the multi-layer design derive from the division of system primitives and system commands. Frequently, primitives remain unchanged while commands need to be modified or extended. By separating the parts, one can maintain a stable and simple layer for primitives while developing an evolving and complex layer for commands. Additionally, since the commands are described in the *meta*-language, users may

add their own commands without resorting to serious programming, assuming the *meta*-language is reasonably easy to use. The highest layer is also insulated from possible changes in the underlying layers and can handle primarily with human factors issues. Thus, the multi-layer design can lead to an extensible and modular system.

The main drawback of the multi-layer design is its complexity. In addition to the modeling system primitives and commands, one must implement yet another system for the *meta*-language, which involves designing and interpreting another language. Complexity has an impact on maintenance as well. When a bug manifests itself, one must first determine where the cause of the problem lies. Since the system consists of three layers, isolating the problem may not be an easy task. On the other hand, once the offending layer has been identified, debugging can proceed rapidly since one need not worry about other layers.

A secondary drawback to the multi-layer design is potential inefficiency. If the user may add or modify commands at run-time, **MIRAGE** must be able to parse and execute system commands at run-time, *i.e.*, be an interpreter for the *meta*-language; and interpreters are potentially very inefficient.

5.2.3. Hybrid Design

To create an extensible **and** efficient system, one might use a mixture of the single-layer and multi-layer designs. By implementing the most frequently used commands in code while having an extensible command language, the system retains the advantages of both systems.

A hybrid system may be approached starting with either pure system. One can start with a single-layer system and treat unknown commands as user-defined commands

instead of errors; however, one must then implement a *meta*-language for handling these user-defined commands. This added complexity removes several advantages of the single-layer approach. A more promising method is to start with a multi-layer system and migrate frequently used commands into the lowest layer as primitives by either manually recoding or using a *meta*-language compiler.

Regardless of which approach is taken, maintaining a hybrid system requires more effort than maintaining the initial system. Since system structure is no longer well defined, isolating bugs becomes a difficult task. Determining whether system commands or system primitives cause problems does not help much because the two sets are no longer distinct. This is the price one pays in the hybrid system for efficiency and extensibility.

5.3. MIRAGE

The primary emphasis of **MIRAGE**, the molecular modeling interface to **MMWB**, is the display and manipulation of three-dimensional molecular models. The molecular systems displayed by **MIRAGE** may be divided roughly into three types. The first type consists of sets of interacting molecules, for example, an enzyme-substrate complex. The distinguishing characteristic of these systems is that each model only has a single set of atomic coordinates. The second type of molecular systems consists of a set of spatially related compounds, for example, many conformations of a single molecule, or a series of substituted analogs of a single parent compound. The third type of systems consists of a set of temporally related compounds, for example, snapshots of a molecule from a molecular dynamics simulation. These three types of systems differ in display capabilities. For the single-coordinate systems, the emphasis is on displaying and

manipulating models independently, *e.g.*, docking. For spatially-related-compounds systems, the emphasis is comparing and contrasting a single molecule or groups of molecules. For temporally-related-compounds systems, the emphasis is in examining the progression of events in time. These differing display needs can only be met if **MIRAGE** supplies a versatile set of system primitives.

While the three types of molecular systems have unique system requirements, they also share an extensive set of common operators. For example, focussing on a small region of the models such as an active site, and labeling specific atoms are both operations that may be required regardless of which type of system is being displayed. In general, accessing and modifying molecular data are generically useful operations. Another set of operations that is generally useful is those dealing with external applications, *e.g.*, sending an atomic coordinate set to an energy minimizer.

The remainder of this section discusses how the functionality provided by **MIRAGE** is implemented by dividing operations into low-level primitives, combining primitives into powerful system commands, and providing easy access to these commands to the user.

5.3.1. System Primitives Layer

Primitives provide all the functionality that an interface can present to users. The set of primitives should be as complete and orthogonal as possible. Completeness guarantees that all semantically permissible commands can actually be performed. Orthogonality makes implementation of commands easier because the combination of primitives into commands is simplified. With these guidelines, the primitives of **MIRAGE** may be divided into three groups: molecular, external, and graphical

operations.

5.3.1.1. Molecular Primitives

Molecular primitives handle accessing and modifying molecular data. Operators and their functions are listed in Table 6.

open	Fetch the molecular data associated with a model
close	Release the molecular data associated with a model
group	Assign a label to a group of atoms
ungroup	Dissolve the group
get	Fetch the value of an attribute for an atom
set	Define the value of an attribute for an atom

Table 6 – List of molecular primitives

While the set of primitives is small, the power provided by these simple operations is sufficient for most purposes. More primitives may be added for efficiency purposes, *e.g.*, reduction operators such as center-of-mass computation.

5.3.1.2. External Primitives

External primitives deal with entities outside of **MIRAGE**. These entities can be divided into two groups: applications that understand some aspect of molecular modeling, and other support programs. The external primitive operators distinguishes between them because the molecular data requirements of the two classes of applications are very different. Table 7 lists the operators and functionalities.

call	Initiate another modeling application
start	Initiate a support program
wait	Wait for an application to terminate

Table 7 – List of external primitives

The communications between applications was discussed in

Chapter 2 and uses simple network protocols. Thus, **MIRAGE** can call any external programs that uses the generic **MMWB** communications protocol. Currently, the generic protocol is quite limited, however, with suitable development efforts, a more complete and powerful protocol set may be designed.

5.3.1.3. Graphics Primitives

Graphics primitives converts all the molecular data and external communications into forms that users can understand. This includes displaying molecular models as graphical objects, showing output from external applications as either text or graphs, and generally maintaining the appearance of the output device. Since **MIRAGE** is designed for powerful three-dimensional graphics workstations, the list of graphics primitives is quite long, and is shown in Table 8.

graph	Construct the graphical equivalent of the molecular data
draw	Show the graphical constructs
group	Group graphics constructs into a single manipulation unit
ungroup	Dissolve group into individual component
select	Add graphics construct to current manipulation unit
deselect	Remove graphics construct from current manipulation unit
center	Define center of rotation
rotate	Rotate current manipulation unit
translate	Translate current manipulation unit
window	Compute a view which shows all visible graphics constructs
scale	Scale the entire view
clip	Move the view clipping planes
match	Superimpose two graphics constructs or groups
save	Save the current graphics transformations and view
restore	Restore a previously saved view
config	Set graphics configuration parameter
inform	Display text information message
print	Make a hardcopy of the current view

Table 8 – List of graphics primitives

Table 8 is a minimal set of graphics operators. Many others may be added to access the specific capabilities of individual workstations. However, with this minimal set of operators, one can provide nearly all the desired functionality needed in a molecular modeling system.

5.3.2. System Commands Layer

While the primitive layer provides the required functionality of the interface, it does so by providing many independent and orthogonal operators. However, many common user activities consist of executing several primitives sequentially. For example, when a user wants to display a label for an atom, the primitive operations needed are editing the data associated with the atom, remaking the graphics construct, and redisplaying the graphics construct. The sequence of primitives always remains the same and, hence, may be grouped into a system command.

Grouping primitives into commands may either be done at system compilation time or during execution time. When done at compile time, the system command is converted into machine code and cannot be altered further. When done at execution time, the system command is constructed at program start-up time. The advantage of the former is execution speed; the advantage of the latter is flexibility. In either case, the grouping of primitives may be done using a *meta*-language.

5.3.2.1. *meta*-language

A *meta*-language is a language which describes another language. In this case, the **MIRAGE** *meta*-language describes the system commands available to users in terms of

system primitives. Two goals in designing the MIRAGE *meta*-language are completeness and simplicity. Completeness is always desirable in any language, and especially in a programming language. Simplicity is also a criterion because system commands may be implemented by users unfamiliar with computer programming; therefore, a simple language where chances for mistakes are minimized is preferred.

The MIRAGE *meta*-language must be capable of describing both the syntax and the semantics of a system command. This is similar to function definition in imperative programming languages, where the function arguments are listed, followed by the actions to be executed. A sample description of the *color* command is shown in Figure 6.[7]

```
color
        c:      string;                 # color name or number
        al:     optional atom_list;     # atoms to color
{
        a:      atom;
        num:    integer;

        num = color_number(c);
        iterate(a, al) {
                set(a, "color", num);
        }
        graph();
        redraw();
}
```

Figure 6 – *meta*-language description of *color* command

Figure 6 shows that the *color* command takes two arguments, a color (of type *string*), and a list of atoms (of type *atom_list*). The actions associated with the command are to convert the color string into a color number, set the *color* attribute of the selected atoms to the color number, rebuild and redisplay the graphics constructs. This simple example shows that the *meta*-language is a programming language, since it contains data types,

[7] A small grammar for parsing the sample *meta*-language appears in Appendix B.

execution statements, and branching operators. It also contains domain-specific features, such as the *atom_list* data type, and primitive operators such as *set*.

5.3.3. Interpretation Layer

The interpretation layer translates user actions into system command or primitive invocation. The four primary responsibilities of the interpretation layer is presentation (*e.g.*, displaying text input area and control panel), input handling (*e.g.*, processing keystrokes and mouse clicks), user feedback (*e.g.*, echo keystrokes), and invoking system commands and primitives. The first three areas together provide the **look and feel** of **MIRAGE**, while the latter provides the modeling capabilities.

Much has been written about the **look and feel** of user interfaces. The style of interface that **MIRAGE** uses is a combination of the classical keyboard interface and the point-and-click interface first developed for the Xerox Star workstation. A textual area is allocated for displaying keyboard input. Mouse clicks over the background generate pop-up menus whose contents are dependent on the current content of the text input area. Items from the menus may be selected and are automatically entered into the text input area. Mouse clicks over an icon representing molecules or applications will generate pop-up menus which are appropriate for the chosen item, *e.g.*, clicking on an external application icon generates a menu with items **invoke, describe**, and **edit parameters**. While this interface does not have the theoretical background behind some other interface styles, it has the great advantage of familiarity to current users of molecular modeling software.

Some user actions may be mapped to system commands or primitives immediately. For example, clicking the mouse over an area of the screen representing a joystick may

be translated to invoking a molecule rotation primitive. Other user actions affect the state of the interface. For example, a keystroke may simply add another character to the text input area; however, if the keystroke happens to be **return**, then the content of the input area is then interpreted as a system command, and processed accordingly. Thus users may access all the functionality provided by the two lower layers through the interpretation layer.

5.3.4. Summary of MIRAGE

Two aspects of **MIRAGE** need more work. First, the basic **MMWB** communications protocol must be expanded so **MIRAGE** may communicate with a large set of applications. Second, further studies need to be done on the **look and feel** provided by **MIRAGE** and how it may be improved. With these areas enhanced, **MIRAGE** should provides a good molecular modeling interface to **MMWB**.

5.4. Conclusion

Table 9 highlights the features of the different approaches to constructing user interfaces.

Design Consideration	Framework Type		
	Single-layer	Multi-layer	Hybrid
Extensibility	Extensible only by modifying code	Extensible at run-time	Extensible at run-time
Efficiency	Very efficient	Possibly inefficient	Very efficient for internal commands. External commands may be inefficient.
Development	Need to develop modeling system	Need to develop both a language system and a modeling system	Need to develop a language system and a modeling systems
Maintenance	Easy	Moderate	Difficult

Table 9 – Features of user interface programming frameworks

The single-layer approach has been used successfully in existing molecular modeling systems. However, because **MMWB** will deal with a wide spectrum of problems, an extensible system is highly desirable. An extensible system which can define not only the syntax but also the semantics of new commands is particularly useful since it would place the power of customization in the hands of users.

The multi-layer approach is the most elegant of the three frameworks. It provides the desired extensibility while supplying a uniform and well defined interface between system primitives and commands. If the potential problem of efficiency arises, one may sacrifice elegance and settle with the hybrid approach by reimplementing commands as primitives. By trading simplicity for efficiency, one may achieve the desired balance of speed and functionality.

MIRAGE, the molecular modeling interface to **MMWB**, is designed as a multi-layered interface. The primitives are divided into molecular, external, and graphical

operations. These primitives are combined into commands using a simple *meta*-language. The efficiency of **MIRAGE** cannot be determined until a prototype interface is implemented. Decisions on which commands should be migrated into the primitive layer will be made at that time. The interpretation layer of **MIRAGE** currently uses both key-board and menu input. This approach is only the first step in making the interface user-friendly. A more detailed study of the user population once **MIRAGE** is implemented should yield trends which will help determine whether a more complex interface is required and what features are desirable.

Chapter 6

Future Directions

6.1. Introduction

The MMWB environment addresses three problems of molecular modeling in a network environment:

- Communications among services
- Data exchange format among services
- User access to services

By defining standards in these three areas, MMWB provides the foundation for an extensible modeling environment. Although these three areas form the cornerstone of MMWB, several other issues need to be resolved to provide a complete system. Unfinished business includes:

- Load balancing
- Administration
- External data exchange format

Future extensions might include:

- High level data organization
- Multiple user interfaces
- Consistent treatment of data

6.2. Unfinished Business

Issues that deal with low level areas such as communications are considered unfinished business because they are affected by the design of the areas addressed by MMWB. None of the issues listed above are critical to the operation of MMWB. However, they should be resolved to complete the low level design.

Load balancing is the attempt to distribute the computing load evenly across hosts

on the network. There have been many load balancing schemes proposed for general computing environments, for example, **maitre d'** [4]. Most of the schemes involve keeping track of the load average on the participating hosts and sending compute bound jobs to the least loaded host. **MMWB** can easily join such schemes by checking for load averages either at the user interface or the dispatcher. However, the user should retain the authority to specify service execution on a designated host regardless of computing load.

Administration of **MMWB** includes publishing approved communications protocols, table formats, and object formats; maintaining lists of services provided by various hosts (*i.e.*, keeping the dispatcher lists of services up to date); and resolving disputes among privately developed services. The first task, publishing standards, is actually quite simple because the administrator can require that the standard description be written by the implementor and be in publishable form before approving it. The second task, maintaining lists, is more tedious as different hosts may support different services; however, the actual work involved, editing configuration files, is not great since adding and deleting services are infrequent events. The third task, resolving disputes, is mostly policy selection and enforcement.

External data exchange formats are needed to communicate with sites that do not use the **MMWB** internal data format. Formats from libraries such as the Protein Data Bank [1,3] are simple transformations of the **MMWB** object and table format. Conversion programs need to be written that converts to and from each of the required external formats.

6.3. Future Extensions

MMWB provides the foundation of a modeling environment. The facilities that are constructed on top of this foundation defines the structure of the entire system. By defining standards for higher level structure, the modeling environment gains a greater degree of consistency.

Data are currently organized into objects. By organizing objects into even higher-level aggregates, the data management subsystem can reduce the amount of detail with which users need to contend. For example, if objects are organized into projects, then the user interface only needs to present objects in the project that the user has selected. Objects from other projects are completely hidden under the project abstraction.

MMWB describes why many user interfaces may be required to address the different aspects of molecular modeling. However, these interfaces should be consistent at some level so that learning time for users is reduced once they master one of the interfaces. Similarly for privately developed computational services, the invocation method should be consistent at some level so the user interfaces can invoke new services without explicit knowledge about them. The consistency may be achieved by using a **metaphor**, which maps a computer representation of information to a real-world situation, such as working at a chemistry workbench. This consistency should make the system easier to learn and appear more integrated to users.

6.4. Summary

MMWB provides a flexible and extensible environment for molecular modeling using a network of computers. The computing tasks are divided into **services** which are distributed across the network to take advantage of the different capabilities of different

hosts. The services communicate with each other using standardized **communications protocols**. A single copy of data is stored and managed on one machine by a **data manager** and all services share the data by communicating with the data manager via the **object file system**. A user accesses the computing resources via a **user interface** which presents data to the user and helps him manage the plethora of data and processes.

Further development using higher level data aggregates such as **projects** will further improve the quality of the user interface. Using **metaphors** will improve the consistency of all elements of **MMWB** including compute servers and user interfaces. The resulting product will be a powerful tool for investigating the structure and function of macromolecules.

References

1. Abola, E. E., F. C. Bernstein, S. H. Bryant, T. F. Koetzle, and J. Weng, *Crystallographic Databases - Information Content, Software Systems, Scientific Applications*, Bonn/Cambridge/Chester, 1987.

2. Accetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Conference Proceedings*, USENIX Association, Summer 1986.

3. Bernstein, F. C., T. F. Koetzle, G. J. B. Williams, E. F. Meyer, Jr, M. D. Brice, J. R. Rodgers, O. Kennard, T. Shimanouchi, and M. Tasumi, *J Mol Biol*, p. 535-542, 1977.

4. Bershad, B., "Load Balancing with Maitr'd," *UCB/Computer Science Department 85/276*, Computer Science Division (EECS), University of California, Berkeley, Berkeley, California, 1987.

5. Calingaert, P., *Operating System Elements, A User Perspective*, Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1982.

6. Comer, D., *Internetworking with TCP/IP, Principles, Protocols, and Architecture*, Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1988.

7. Date, C. J., *An Introduction to Database Systems*, Addison-Wesley Publishing Company, 1986.

8. "NIH Sponsored Prophet System," Division of Research Resource, *Proc Natl Comput Cons Exposition*, 43: 457, 1974.

9. Ferrin, T. E., C. C. Huang, L. E. Jarvis, and R. Langridge, "Molecular Interactive Display and Simulation (MIDAS)," *J Mol Graphics*, p. 13-27, 1988.

10. Horton, M., "Standard for Interchange of USENET Messages," *RFC 1036*, Network Information Center, SRI International, Menlo Park, California, December 1987.

11. *Internet Protocol Implementation Guide*, Network Information Center, SRI International, Menlo Park, California, August 1982.

12. Johnson, S. C., "Yacc: Yet Another Compiler-Compiler," in: *Unix Programmer's Supplementary Documents*, vol. 1, Usenix Association, November 1986.

13. Knuth, D. E., *The Art of Computer Programming*, Addison-Wesley Publishing Company, 1968.

14. Korth, H. and A. Silberschatz, *Database System Concepts*, McGraw-Hill, Inc, 1986.

15. Mockapetris, P., "Domain Names - Implementation and Specification," *RFC 1035*, Network Information Center, SRI International, Menlo Park, California, November 1987.

16. Padlipsky, M. A., *The Elements of Networking Style*, Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1985.

17. "A Protein Sequence/Structure Database," Protein Engineering Club Database Group, *Nature*, 335: 745-746, 20 October 1988.

18. Rashid, R., "Threads of a New System," *Unix Review*, August 1986.

19. *Readings in Human-Computer Interaction: A Multidisciplinary Approach*, Morgan Kaufmann Publishers, Inc, Los Altos, California, 1989.

20. Steiner, Neuman, and Schiller, ''Kerberos: An Authentication Service for Open Network Systems,'' *Usenix Proceedings*, Winter 1988.

21. ''Networking on the Sun Workstation,'' Sun Microsystems, Inc, *Part No: 800-1324-03, Revision B*, February 1986.

22. ''NFS: Network File System Protocol Specification,'' Sun Microsystems, Inc, *RFC 1094*, Network Information Center, SRI International, Menlo Park, California, March 1989.

23. Tanenbaum, A. S., *Computer Networks*, Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1981.

24. Tufte, E. R., *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, Connecticut, 1983.

25. Weiner, P. K. and P. A. Kollman, ''AMBER: Assisted Model Building with Energy Refinement. A General Program for Modeling Molecules and their Interactions,'' *J Comput Chem*, 2(3): 287-303, 1981.

26. Weissman, C., *Lisp 1.5 Primer*, Dickenson Publising Company, Inc, Belmont, California, 1967.

27. ''Internet Transport Protocols,'' Xerox Corporation, *XSIS-028112*, Stamford, Connecticut, 1981.

Glossary

aggregate

A collection of **objects**.

class, table

See **table class**.

conversation

A set of processes acting cooperatively to complete task.

data manager

A data management service which deals with data in the form of tables.

dispatcher

A program which executes on a single host and provides server locations to clients requesting service.

DM See **data manager**.

***meta*-language**

A language for describing a user interface system command in terms of the system primitives.

MIRAGE

The molecular graphics user interface to **MMWB**.

MMWB

A modern molecular modeling software environment.

object

A collection of information associated with an entity or relationship.

object file system

A system for managing objects and their contents.

object type

A characteristic which distinguishes a set of objects from other objects.

project

A special type of **aggregate** for grouping **objects** from the same research project.

table

A data structure for storing information in rows and columns.

table class

A characteristic which distinguishes a set of tables from other tables.

well known location

A network contact location which is defined *a priori* for all applications.

Appendix A

Dispatcher Interface

Network Protocol: Commands and Responses

SERV service_id

SERV is the request-for-server command. The single argument to the command is a service identifier, consisting of an string of alphanumeric characters. The possible responses to the **SERV** command are:

202 Service started. Contact point follows.
404 Service is not provided.
405 Service is currently unavailable.

DUMP type

DUMP is the print-server-list command. The *type* argument may be either **services** or **servers**. If it is **services**, the response to the command should be the list of available services. If it is **servers**, the response should be the list of active servers. The possible responses to this command are:

202 ASCII data follows
202 List of servers:
202 List of services:
204 No active servers.
204 No services available.

Functional Interface: Library Routines

```
typedef   void        *DISPATCHER;
extern int            dispatcher_errno;
```

```
DISPATCHER
dispatcher_open(host, acct, id)
char        *host;     /* Host where service is requested */
char        *acct;     /* Remote account to use */
char        *id;       /* Conversation identifier */
```

This function is used to establish a connection to the dispatcher on a remote host. The low-level handshaking is done by this routine. The return value is a pointer to the dispatcher if the handshaking succeeds, NULL otherwise.

```
int
dispatcher_close(dp)
DISPATCHER            dp;        /* Remote dispatcher */
```

This function is used to terminate an established connection to a remote dispatcher. The return value is 0 on success and -1 on failure.

```
int
dispatcher_request(dp, service)
DISPATCHER            dp;        /* Remote dispatcher */
char        *service; /* Name of requested service */
```

This function is used to request the designated service from a remote host. The return value is a file descriptor to a server on success, -1 otherwise.

```
int
dispatcher_call(host, acct, id, service)
char        *host;
char        *acct;
char        *id;
char        *service;
```

This function is the combination of the three previous functions and is equivalent to calling **dispatcher_open**, **dispatcher_request**, and **dispatcher_close** in sequence. The return value is a file descriptor to the remote server if all three called functions are successful, -1 otherwise.

```
void
dispatcher_perror(fp, s)
FILE      *fp;                    /* Where to send error */
char      *s;                     /* Error message prefix */
```

This function is used to print an error message based on the current value of the error variable *dispatcher_errno*. There is no return value.

Appendix B

meta-language Grammar

The following YACC program may be used to parse a subset of the *meta*-language.

```
%token IDENT CONSTANT OPTIONAL ITERATE
%%
file            :       file command
                |
                ;
command         :       head body
                ;
head            :       IDENT decl_list
                ;
decl_list       :       decl_list decl
                |
                ;
decl            :       IDENT ':' optional type ';'
                ;
optional        :       OPTIONAL
                |
                ;
type            :       IDENT
                ;
body            :       '{' decl_list stmt_list '}'
                ;
stmt_list       :       stmt_list statement
                |       statement
                ;
statement       :       primitive ';'
                |       assignment
                |       iterate
                ;
primitive       :       IDENT '(' arg_list ')'
                |       IDENT '(' ')'
                ;
arg_list        :       arg_list ',' arg
                |       arg
                ;
arg             :       IDENT
                |       CONSTANT
                ;
assignment      :       IDENT '=' expression ';'
                ;
expression      :       expression op term
```

```
                |       term
                ;
term            :        primitive
                |       IDENT
                |       CONSTANT
                ;
op              :        '+'
                |       '-'
                |       '*'
                |       '/'
                |       '|'
                |       '&'
                ;
iterate         :        it_head statement
                |       it_head body
                ;
it_head         :        ITERATE '(' IDENT ',' IDENT ')'
                ;
%%
#include "lex.yy.c"

yyerror(s)
char    *s;
{
        fprintf(stderr, "Line %d: %s\n", yylineno, s);
}
```

The following LEX program may be used as a simple lexical analyzer.

```
%%
optional                {       return OPTIONAL;        }
iterate                 {       return ITERATE;         }
[a-zA-Z_][a-zA-Z0-9_]*  {       return IDENT;           }
[0-9]*                  {       return CONSTANT;        }
\"[^"]*\"               {       return CONSTANT;        }
[ \t]*                  {       ;                       }
\n                      {       ;                       }
%%
```

Appendix C

Prototype Implementation

Prototypes of both the data manager and the functional interface have been implemented. These prototypes make certain assumptions, such as all tables in a data set will be accessed in a session, which are detrimental to performance. Since data access will play an important role in MMWB, these prototypes should be replaced with new and efficient versions. The prototype code listed here, while complete in functionality, is only used for proof-of-concept purposes.

dm/attribute.c

```
/*
 * $Header: attribute.c,v 1.7 89/03/02 22:01:05 pett Exp $
 * $Log:            attribute.c,v $
 * Revision 1.7  89/03/02  22:01:05  pett
 * declared emalloc/erealloc to return void *
 *
 * Revision 1.6  88/09/28  10:43:08  conrad
 * Implement multiple groups if NGROUPS is defined
 *
 * Revision 1.5  88/08/08  09:46:02  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.4  88/08/04  16:26:10  conrad
 * Support data type STRING and TABLE
 *
 * Revision 1.3  88/07/29  14:59:13  conrad
 * Update comments
 *
 * Revision 1.2  88/07/27  20:21:45  conrad
 * Use reply parameter instead of hardwired numbers
 * Describe user with account name instead of uid/gid
 *
 * Revision 1.1  88/07/25  13:15:01  conrad
 * Initial revision
 *
 */

#include "table.h"
#include "dataset.h"
#include "error.h"
#include "reply.h"
#include <pwd.h>

/*
 * attr_release:
 *      Release a list of attributes
 */
void
attr_release(ap)
register ATTR   *ap;
{
        register ATTR   *next;

        while (ap != NULL) {
                next = ap->next;
                attr_free(ap);
                ap = next;
        }
}

/*
 * attr_free:
 *      Release an entire attribute structure
 */
void
attr_free(ap)
register ATTR   *ap;
{
        dblock_release(ap->dblock);
        (void) free(ap->name);
        (void) lock_free(ap->lock);
        (void) free((char *) ap);
}

/*
 * attr_new:
 *      Create a new attribute
 */
int
attr_new(clp, tname, aname, dtype, dsize)
CLIENT  *clp;
char    *tname, *aname;
int     dtype, dsize;
{
        register TABLE  *tp;
        register ATTR   *ap;
        struct passwd   *pp;
        char            *copy_string();
        extern void     *emalloc();
        extern time_t   time();

        /*
         * See if we can store this data type
         */
        if (dt_supported(dtype) < 0)
                return ATTR_UNSUPPORTED;

        /*
         * Make sure the table exists
         */
        tp = tbl_find(tname);
        if (tp == NULL || tp->deleted)
                return TBL_NOSUCH;
```

```
	/*
	 * Make sure the client has a write lock on the table
	 */
	if (lock_mode(tp->lock, clp) != LOCK_WRITE)
		return TBL_NOPERM;

	/*
	 * Make sure the attribute does not already exist
	 */
	for (ap = tp->attr; ap != NULL; ap = ap->next)
		if (strcmp(ap->name, aname) == 0)
			break;
	if (ap != NULL)
		return ATTR_EXISTS;

	/*
	 * Create the new attribute
	 */
	ap = (ATTR *) emalloc(sizeof (ATTR));
	ap->next = tp->attr;
	tp->attr = ap;
	ap->lock = lock_find(LOCK_ATTR, (void *) ap);
	ap->table = tp;
	ap->modified = FALSE;
	ap->dblock = NULL;
	ap->name = copy_string(aname);
	ap->ncell = -1;
	ap->ndata = 0;
	ap->uid = clp->uid;
#ifdef NGROUPS
	ap->gid = clp->groups[0];		/* This is not used anywhere */
#else
	ap->gid = clp->gid;
#endif
	pp = getpwuid((int) ap->uid);
	if (pp == NULL)
		ap->owner = "nobody";
	else
		ap->owner = copy_string(pp->pw_name);
	ap->mtime = ap->atime = time((time_t *) 0);
	ap->dtype = dtype;
	ap->dsize = dsize;
	if (ap->dtype == DM_S_STRING || ap->dtype == DM_S_TABLE)
		str_init(ap);

	/*
	 * Mark the table as modified so it will be saved to disk
	 */
	tp->modified = TRUE;
	tp->mtime = ap->mtime;
	tp->nattr++;
	return ATTR_SUCCESS;
}

/*
 * attr_delete:
 *	Delete an attribute
 */
int
attr_delete(clp, tname, aname)
CLIENT	*clp;
char	*tname, *aname;
{
	register TABLE	*tp;
	register ATTR	*ap, *prev;

	/*
	 * Find locked table
	 */
	tp = tbl_find(tname);
	if (tp == NULL || tp->deleted)
		return TBL_NOSUCH;
	if (lock_mode(tp->lock, clp) != LOCK_WRITE)
		return TBL_NOPERM;

	/*
	 * Find target attribute
	 */
	prev = NULL;
	for (ap = tp->attr; ap != NULL; ap = ap->next) {
		if (strcmp(ap->name, aname) == 0)
			break;
		prev = ap;
	}
	if (ap == NULL)
		return ATTR_NOSUCH;

	/*
	 * Remove attribute from list
	 */
	if (prev == NULL)
		tp->attr = ap->next;
	else
```

```
			prev->next = ap->next;

		/*
		 * Release data associated with attribute
		 */
		attr_free(ap);

		/*
		 * Update table information
		 */
		tp->modified = TRUE;
		tp->mtime = ap->mtime;
		tp->nattr--;
		return ATTR_SUCCESS;
}

/*
 * attr_list:
 *		List the attribute names for the given client
 */
int
attr_list(clp, tname, atype)
CLIENT	*clp;
char	*tname;
int	atype;
{
		register TABLE	*tp;
		register ATTR	*ap;
		register int	max;

		/*
		 * Find table
		 */
		tp = tbl_find(tname);
		if (tp == NULL || tp->deleted)
			return TBL_NOSUCH;
		if (lock_mode(tp->lock, clp) == LOCK_NONE)
			return TBL_NOPERM;

		/*
		 * List them to the client
		 */
		reply(CL_FILENO(clp), RS_ACCEPT, RT_DM, RG_OKAY, "List of attributes
		reply_begin_ASCII(CL_FILENO(clp));
		max = inst_max(tp) + 1;
		for (ap = tp->attr; ap != NULL; ap = ap->next) {
			if (ap->ndata == 0 && atype != DM_A_EXIST)
				continue;
			if (ap->ndata < max && atype == DM_A_FULL)
				continue;
			reply_ASCII(CL_FILENO(clp), ap->name);
		}
		reply_end_ASCII(CL_FILENO(clp));
		return ATTR_SUCCESS;
}

/*
 * attr_describe:
 *		Describe an attribute
 */
int
attr_describe(clp, tname, aname)
CLIENT	*clp;
char	*tname, *aname;
{
		register TABLE	*tp;
		register ATTR	*ap;
		char		buf[BUFLEN];

		/*
		 * Find the table
		 */
		tp = tbl_find(tname);
		if (tp == NULL || tp->deleted)
			return TBL_NOSUCH;

		/*
		 * Check the permission
		 */
		if ((ds_permission(clp) & DS_PERM_READ) == 0)
			return TBL_NOPERM;

		/*
		 * Find the attribute
		 */
		for (ap = tp->attr; ap != NULL; ap = ap->next)
			if (strcmp(ap->name, aname) == 0)
				break;
		if (ap == NULL)
			return ATTR_NOSUCH;

		/*
		 * Send the descriptions
		 */
```

```c
	reply(CL_FILENO(clp), RS_ACCEPT, RT_DM, RG_OKAY, "Description foll
	reply_begin_ASCII(CL_FILENO(clp));

	/* Send TYPE keyword */
	(void) sprintf(buf, "type %d %d", ap->dtype, ap->dsize);
	reply_ASCII(CL_FILENO(clp), buf);
	/* Send INSTANCE keyword */
	(void) sprintf(buf, "instances %d %d", ap->ncell, ap->ndata);
	reply_ASCII(CL_FILENO(clp), buf);
	/* Send OWNER keyword */
	(void) sprintf(buf, "owner %s", ap->owner);
	reply_ASCII(CL_FILENO(clp), buf);
	/* Send TIME keyword */
	(void) sprintf(buf, "time %ld %ld", ap->mtime, ap->atime);
	reply_ASCII(CL_FILENO(clp), buf);

	reply_end_ASCII(CL_FILENO(clp));

	/*
	 * Now update the access time of the attribute
	 */
	ap->atime = time((time_t *) 0);
	return ATTR_SUCCESS;
}

/*
 * attr_lock:
 *	Lock an attribute
 */
int
attr_lock(clp, tname, aname, mode, key)
CLIENT	*clp;
char	*tname, *aname;
int	mode;
long	*key;
{
	register TABLE	*tp;
	register ATTR	*ap;

	tp = tbl_find(tname);
	if (tp == NULL || tp->deleted)
		return TBL_NOSUCH;

	for (ap = tp->attr; ap != NULL; ap = ap->next)
		if (strcmp(ap->name, aname) == 0)
			break;
	if (ap == NULL)
		return ATTR_NOSUCH;

	if (mode == LOCK_WRITE
	&& (ds_permission(clp) & DS_PERM_WRITE) == 0)
		return ATTR_NOPERM;

	if ((*key = lock_get(ap->lock, mode, *key, clp)) < 0)
		return ATTR_LOCKWAIT;
	return ATTR_SUCCESS;
}

/*
 * attr_lockmode:
 *	Change the mode of a lock
 */
int
attr_lockmode(clp, tname, aname, mode, key)
CLIENT	*clp;
char	*tname, *aname;
int	mode;
long	*key;
{
	register TABLE	*tp;
	register ATTR	*ap;

	tp = tbl_find(tname);
	if (tp == NULL || tp->deleted)
		return TBL_NOSUCH;

	for (ap = tp->attr; ap != NULL; ap = ap->next)
		if (strcmp(ap->name, aname) == 0)
			break;
	if (ap == NULL)
		return ATTR_NOSUCH;

	switch (lock_chmod(ap->lock, mode, key, clp)) {
	 case -1:
		return ATTR_NOPERM;
	 case 0:
		return ATTR_SUCCESS;
	 case 1:
		return ATTR_LOCKWAIT;
	 default:
		error_internal("attr_lockmode failed");
		/* NOTREACHED */
	}
}
```

dm/attribute.c

```
/*
 * attr_unlock:
 *		Unlock an attribute
 */
int
attr_unlock(clp, tname, aname, key)
CLIENT	*clp;
char	*tname, *aname;
long	key;
{
	register TABLE	*tp;
	register ATTR	*ap;

	tp = tbl_find(tname);
	if (tp == NULL || tp->deleted)
		return TBL_NOSUCH;

	for (ap = tp->attr; ap != NULL; ap = ap->next)
		if (strcmp(ap->name, aname) == 0)
			break;
	if (ap == NULL)
		return ATTR_NOSUCH;

	if (lock_release(ap->lock, key, clp) < 0)
		return ATTR_NOPERM;
	return ATTR_SUCCESS;
}

/*
 * attr_sleep:
 *		Go to sleep waiting for the lock of this attribute
 */
void
attr_sleep(clp, tname, aname, mode)
CLIENT	*clp;
char	*tname, *aname;
int	mode;
{
	register TABLE	*tp;
	register ATTR	*ap;

	tp = tbl_find(tname);
	if (tp == NULL || tp->deleted) {
		error_internal("attr_sleep failed(1)");
		/* NOTREACHED */
	}
	for (ap = tp->attr; ap != NULL; ap = ap->next)
		if (strcmp(ap->name, aname) == 0)
			break;
	if (ap == NULL) {
		error_internal("attr_sleep failed(2)");
		/* NOTREACHED */
	}
	cl_sleep(ap->lock, mode, clp);
}
```

dm/client.c

```
/*
 * $Header: client.c,v 1.5 89/03/02 20:59:02 pett Exp $
 * $Log:          client.c,v $
 * Revision 1.5  89/03/02  20:59:02  pett
 * declared emalloc to return void *
 *
 * Revision 1.4  88/10/04  15:03:12  conrad
 * Handle inputs as true streams with no record boundaries (newlines are
 * no longer expected as the last character of an input buffer)
 *
 * Revision 1.3  88/09/29  14:05:20  conrad
 * Handle input properly (i.e. multiple command in the same buffer now works)
 * EOF by clients waiting for a lock also handled properly now
 *
 * Revision 1.2  88/08/08  09:46:18  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.1  88/07/25  13:15:19  conrad
 * Initial revision
 *
 */

#include "client.h"
#include "lock.h"

CLIENT  *clist = NULL;              /* Nothing initially */

/*
 * cl_new:
 *      Create a a new client
 */
CLIENT *
cl_new(fd)
int     fd;
{
        register CLIENT     *clp;
        extern void         *emalloc();

        clp = (CLIENT *) emalloc(sizeof (CLIENT));
        clp->next = clist;
        clist = clp;
        clp->state = CS_VERS;
        clp->fd = fd;
        clp->indx = 0;
        clp->combuf = NULL;
        return clp;
}

/*
 * cl_free:
 *      Release client
 */
cl_free(target)
CLIENT  *target;
{
        register CLIENT     *clp, *prev;

        prev = NULL;
        for (clp = clist; clp != NULL; clp = clp->next) {
                if (clp == target)
                        break;
                prev = clp;
        }
        if (clp != target)
                return -1;
        if (prev == NULL)
                clist = clp->next;
        else
                prev->next = clp->next;
        lock_remove_client(clp);
        (void) close(clp->fd);
        (void) free((char *) clp);
        return 0;
}

/*
 * cl_input:
 *      Get client to read input
 */
void
cl_input(mask)
fd_set  *mask;
{
        register CLIENT     *clp;
        register int        cnt;

        for (clp = clist; clp != NULL; clp = clp->next) {
                if (!FD_ISSET(CL_FILENO(clp), mask))
                        continue;
                cnt = read(clp->fd, &clp->ibuf[clp->indx], IO_SIZE - clp->indx);
                if (cnt <= 0) {
                        (void) cl_free(clp);
```

```
				return;
			}
			clp->indx += cnt;
			clp->ibuf[clp->indx] = '\0';
			while (cl_process(clp, FALSE) == 0)
				continue;
		}
}

/*
 * cl_sleep:
 *	Get client to temporarily halt processing a command
 */
void
cl_sleep(lp, mode, clp)
LOCK	*lp;
int	mode;
CLIENT	*clp;
{
	clp->state = CS_LOCKWAIT;
	lock_wait(lp, mode, clp, cl_wakeup);
}

/*
 * cl_wakeup:
 *	Get client to resume processing a command
 */
cl_wakeup(lp, mode, clp)
LOCK	*lp;
int	mode;
CLIENT	*clp;
{
#if defined(DEBUG) || defined(lint)
	char	cbuf[BUFLEN], lbuf[BUFLEN];

	if (lock_get(lp, mode, lp->key, clp) < 0) {
		cl_name(clp, cbuf);
		lock_name(lp, lbuf);
		syslog(LOG_ERR, "Client %s woke up without lock %s\n",
			cbuf, lbuf);
	}
	(void) lock_release(lp, lp->key, clp);
#endif
	clp->state = CS_DMCMD;
	while (cl_process(clp, TRUE) == 0)
		continue;
}

/*
 * cl_fdmask:
 *	Construct a select() mask for all clients
 */
void
cl_fdmask(mask)
fd_set	*mask;
{
	register CLIENT	*clp;

	for (clp = clist; clp != NULL; clp = clp->next)
		FD_SET(clp->fd, mask);
}

/*
 * cl_data_wait:
 *	Set DATAWAIT state variables
 */
void
cl_data_wait(clp, dtype, dsize, n, iov, attr, from, to)
CLIENT		*clp;
int		dtype, dsize;
int		n;
struct iovec	*iov;
void		*attr;
int		from, to;
{
	clp->dtype = dtype;
	clp->dsize = dsize;
	clp->niov = n;
	clp->iov = iov;
	clp->attr = attr;
	clp->cfrom = from;
	clp->cto = to;
	clp->state = CS_DATAWAIT;
}

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/*
 * cl_name:
 *	Construct a name for a client
 */
```

dm/client.c

```
void
cl_name(clp, buf)
CLIENT	*clp;
char	*buf;
{
	struct sockaddr_in		sa;
	int				size;
#ifndef ntohs
	unsigned short			ntohs();
#endif

	size = sizeof sa;
	if (getpeername(clp->fd, (struct sockaddr *) &sa, &size) < 0)
		(void) sprintf(buf, "0x%x", clp);
	else
		(void) sprintf(buf, "%u@%s", ntohs(sa.sin_port),
			inet_ntoa(sa.sin_addr));
}
```

dm/client.h

```c
/*
 * $Header: client.h,v 1.5 89/02/23 17:49:35 pett Exp $
 * $Log:           client.h,v $
 * Revision 1.5  89/02/23  17:49:35  pett
 * changed declarations of uids/gids to UIDTYPE/GIDTYPE
 *
 * Revision 1.4  88/10/04  15:04:23  conrad
 * Handle inputs as true streams with no record boundaries (newlines are
 * no longer expected as the last character of an input buffer)
 *
 * Revision 1.3  88/09/28  10:43:19  conrad
 * Implement multiple groups if NGROUPS is defined
 *
 * Revision 1.2  88/08/08  09:46:24  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.1  88/07/25  13:15:19  conrad
 * Initial revision
 *
 */

#ifndef   CLIENT_INCLUDE

#include "config.h"
#include "reply.h"
#include <sys/uio.h>
#include <sys/param.h>
#include <machdep.h>

/*
 * Client states
 */
#define   CS_VERS       0        /* Needs version number */
#define   CS_HELO       1        /* Needs conversation id */
#define   CS_USER       2        /* Needs user id */
#define   CS_MACHINE    3        /* Needs machine type */
#define   CS_DMCMD      4        /* Needs data manager command */
#define   CS_LOCKWAIT   5        /* Hanging on a lock */
#define   CS_DATAWAIT   6        /* Waiting for data */

/*
 * Maximum number of tokens in a command
 */
#define   COM_MAX_TOKEN7

/*
 * Client data structure
 */
typedef struct client        {
        struct client        *next;
        int                  state;
        UIDTYPE              uid;            /* ID's on this system */
#ifdef NGROUPS
        int                  ngroups;
        GIDTYPE              groups[NGROUPS];
#else
        GIDTYPE              gid;
#endif
        int                  fd;             /* File descriptor to client */
        int                  data_mode;

        char                 *combuf;        /* Command parsing variables*/
        char                 *args[COM_MAX_TOKEN];
        int                  nargs;

        int                  indx;           /* I/O variables */
        char                 ibuf[BUFLEN];

        int                  dtype;          /* DATAWAIT state variables */
        int                  dsize;
        int                  niov;
        struct iovec         *iov;
        void                 *attr;
        int                  cfrom, cto;
}       CLIENT;

/*
 * Package variables
 */
extern CLIENT     *clist;

/*
 * Macros
 */
#define   CL_FILENO(clp)         (clp)->fd
#define   CL_DATAMODE(clp)       (clp)->data_mode
#define   CL_DTYPE(clp)          (clp)->dtype
#define   CL_DSIZE(clp)          (clp)->dsize
#define   CL_IOVCNT(clp)         (clp)->niov
#define   CL_IOV(clp)            (clp)->iov
#define   CL_ATTRIBUTE(clp)      (clp)->attr
#define   CL_FROM(clp)           (clp)->cfrom
#define   CL_TO(clp)             (clp)->cto
```

dm/client.h

```
#define   CL_BUF(clp)              (clp)->ibuf
#define   CL_BUFSIZ(clp)           (sizeof (clp)->ibuf)
#define   CL_BUFCNT(clp)           (clp)->indx

/*
 * List of access functions
 */
CLIENT    *cl_new();                    /* Create new client */
int       cl_free();                    /* Release old client */
void      cl_input();                   /* Get clients to read input */
int       cl_parse();                   /* Get client to parse command */
void      cl_sleep();                   /* Get client to halt command */
int       cl_wakeup();                  /* Get client to resume command */
void      cl_fdmask();                  /* Construct select() mask */
void      cl_name();                    /* Construct client ident */
void      cl_help();                    /* Print help message */
void      cl_data_wait();               /* Set data wait state variables */
void      cl_consume();                 /* Consume input data */

#define   CLIENT_INCLUDE
#endif
```

dm/config.h

```
/*
 * $Header: config.h,v 1.3 88/09/07 14:50:58 conrad Exp $
 * $Log:           config.h,v $
 * Revision 1.3  88/09/07  14:50:58  conrad
 * Include "machdep.h" instead of using ifdef's
 *
 * Revision 1.2  88/08/08  09:46:26  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.1  88/07/25  13:15:20  conrad
 * Initial revision
 *
 */

#ifndef   CONFIG_DM

#include <stdio.h>
#include <syslog.h>
#include <sys/types.h>
#include <machdep.h>

#define   DM_LOGFILE      "/usr/tmp/dm.log"

#define   BPB      8      /* Bits Per Byte */

#define   IO_SIZE  1023
#define   BUFLEN  (IO_SIZE + 1)

#ifndef   TRUE
#define   TRUE     1
#define   FALSE    0
#endif

#define   CONFIG_DM
#endif
```

dm/dataset.c

```
/*
 * $Header: dataset.c,v 1.5 89/02/23 18:17:31 pett Exp $
 * $Log:		dataset.c,v $
 * Revision 1.5  89/02/23  18:17:31  pett
 * changed some variable declarations to UIDTYPE/GIDTYPE
 *
 * Revision 1.4  88/09/28  10:43:21  conrad
 * Implement multiple groups if NGROUPS is defined
 *
 * Revision 1.3  88/08/08  09:46:27  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.2  88/07/29  14:59:25  conrad
 * Add functions for making proper account own dataset files
 *
 * Revision 1.1  88/07/25  13:15:20  conrad
 * Initial revision
 *
 */

#include "client.h"
#include "dataset.h"
#include "reply.h"
#include <string.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/dir.h>

static char		ds_name[DS_NAMESIZE];
static int		ds_fd = -1;
static u_short		ds_mode = 0;
static UIDTYPE		ds_uid;
static GIDTYPE		ds_gid;
static int		ds_remove = FALSE;
static time_t		ds_atime, ds_mtime;

/*
 * ds_open:
 *	Open a data set
 */
int
ds_open(name, clp)
char	*name;
CLIENT	*clp;
{
	register int		len;
	int			fd, n;
	char			hname[BUFLEN];
	char			buf[BUFLEN];
	struct stat		sbuf;
	void			read_tables();
	extern long		lseek();
	extern time_t		time();

	if (ds_fd != -1) {
		/* Data set is already open */
		if (strcmp(name, ds_name) != 0)
			return DS_WRONGFILE;
		switch (ds_permission(clp) & DS_PERM_RW) {
		 case DS_PERM_RW:
			return DS_READWRITE;
		 case DS_PERM_READ:
			return DS_READONLY;
		 default:
			return DS_NOPERM;
		}
	}

	/*
	 * Get the modes and file descriptor
	 */
	if ((ds_fd = open(name, O_RDONLY)) < 0) {
		if (access(name, F_OK) < 0)
			return DS_NOFILE;
		return DS_NOPERM;
	}
	if (fstat(ds_fd, &sbuf) < 0) {
		(void) close(ds_fd);
		ds_fd = -1;
		return DS_NOPERM;
	}
	ds_mode = sbuf.st_mode;
	ds_uid = sbuf.st_uid;
	ds_gid = sbuf.st_gid;
	(void) sprintf(buf, "%s/%s", name, DS_LOCKFILE);
	if ((fd = open(buf, O_RDWR)) < 0) {
		(void) close(ds_fd);
		ds_fd = -1;
		return DS_NOPERM;
	}
	(void) strcpy(ds_name, name);

	/*
```

dm/dataset.c

```
	 * Now actually try to lock the data set
	 * The procedure is to lock the ID-file first,
	 * then lock the directory.  (All locks are exclusive locks.)
	 * If the directory lock fails, another data manager has it
	 * (we hope).  Then we can just look in the ID-file
	 * for the name of the other data manager.  We also
	 * increment a count kept in the ID-file to let the
	 * owning data manager know that there is another
	 * request for service coming.  If the directory
	 * lock succeeds, then we have the data set and we should
	 * write out our id into the ID-file.  In either case,
	 * we release the ID-file before returning.
	 */
	if (flock(fd, LOCK_EX) < 0) {
		(void) close(ds_fd);
		(void) close(fd);
		ds_fd = -1;
		return DS_NOPERM;
	}
	if (flock(ds_fd, LOCK_EX | LOCK_NB) < 0) {
		(void) read(fd, (char *) &n, sizeof n);
		len = read(fd, name, sizeof buf);
		name[len] = '\0';
		n++;
		(void) lseek(fd, 0L, 0);
		(void) write(fd, (char *) &n, sizeof n);
		(void) write(fd, name, len);
		(void) flock(fd, LOCK_UN);
		(void) close(fd);
		(void) close(ds_fd);
		ds_fd = -1;
		return DS_UNAVAIL;
	}
	else {
		n = 0;
		(void) gethostname(hname, sizeof hname);
		(void) sprintf(buf, "%s@%s\n", ds_name, hname);
		(void) write(fd, (char *) &n, sizeof n);
		(void) write(fd, buf, strlen(buf));
	}
	(void) flock(fd, LOCK_UN);
	(void) close(fd);

	/*
	 * Get the dataset info into memory
	 */
	if (ds_visit() < 0) {
		syslog(LOG_ERR, "%s: %m", ds_name);
		return DS_NOPERM;
	}
	read_tables();
	ds_atime = ds_mtime = time((time_t *) NULL);

	/*
	 * Now we return the permissions for this particular user
	 */
	switch (ds_permission(clp) & DS_PERM_RW) {
	 case DS_PERM_RW:
		return DS_READWRITE;
	 case DS_PERM_READ:
		return DS_READONLY;
	 default:
		return DS_NOPERM;
	}
}

/*
 * read_tables:
 *	Scan through the dataset directory and read in all the tables
 */
static
void
read_tables()
{
	DIR		*dirp;
	struct direct	*dp;
	FILE		*fp;
	struct stat	sbuf;
	int		tbl_read();

	if ((dirp = opendir(".")) == NULL)
		return;
	for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp)) {
		if ((fp = fopen(dp->d_name, "r")) == NULL)
			continue;
		if (fstat(fileno(fp), &sbuf) < 0) {
			(void) fclose(fp);
			continue;
		}
		if ((sbuf.st_mode & S_IFMT) != S_IFREG) {
			(void) fclose(fp);
			continue;
		}
		(void) tbl_read(fp, dp->d_name);
```

```
			(void) fclose(fp);
		}
		closedir(dirp);
}

/*
 * ds_create:
 *		Create a data set
 */
int
ds_create(name, clp)
char	*name;
CLIENT	*clp;
{
		int			fd;
		char			buf[sizeof ds_name + 10];
		extern int		errno;

		if (mkdir(name, DS_DEF_MODE) < 0)
			return (errno == EEXIST) ? DS_EXISTS : DS_NOPERM;
		(void) sprintf(buf, "%s/%s", name, DS_LOCKFILE);
		if ((fd = creat(buf, 0664)) < 0) {
			(void) rmdir(name);
			return DS_NOPERM;
		}
		(void) close(fd);
#ifdef NGROUPS
		(void) chown(name, clp->uid, -1);
		(void) chown(buf, clp->uid, -1);
#else
		(void) chown(name, clp->uid, clp->gid);
		(void) chown(buf, clp->uid, clp->gid);
#endif
		return DS_SUCCESS;
}

/*
 * ds_delete:
 *		Delete a data set
 */
int
ds_delete(name, clp)
char	*name;
CLIENT	*clp;
{
		if (strcmp(name, ds_name) != 0)
			return DS_WRONGFILE;

		/*
		 * We can't just delete all this stuff since there may
		 * still be clients using the data.  We must determine
		 * whether we have permission to delete the contents of
		 * the directory and return a value based on that.  We
		 * really ought to figure out whether the directory itself
		 * may be deleted, but that's another story.
		 */
		if ((ds_permission(clp) & DS_PERM_WRITE) == 0)
			return DS_NOPERM;
		ds_remove = TRUE;
		return DS_SUCCESS;
}

/*
 * ds_cleanup:
 *		Close a data set
 */
int
ds_cleanup()
{
		register int		fd;
		int			n;
		char			buf[sizeof ds_name + 20];
		void			tbl_flush();
		extern long		lseek();

		if (ds_fd < 0)
			return DS_SUCCESS;

		/*
		 * Lock the ID-file and check the count within.  If
		 * the count is non-zero, then there are other requests
		 * coming.  Just decrement the count and return/
		 */
		(void) sprintf(buf, "%s/%s", ds_name, DS_LOCKFILE);
		if ((fd = open(buf, O_RDWR)) >= 0) {
			(void) flock(fd, LOCK_EX);
			(void) read(fd, (char *) &n, sizeof n);
			if (n-- > 0) {
				(void) lseek(fd, 0L, 0);
				(void) write(fd, (char *) &n, sizeof n);
			}
			(void) flock(fd, LOCK_UN);
			(void) close(fd);
			if (n >= 0)
				return DS_MORE;
```

```
	}

	/*
	 * Clear out this directory
	 */
	if (ds_remove) {
		(void) sprintf(buf, "/bin/rm -rf %s", ds_name);
		(void) system(buf);
	}
	else
		tbl_flush();
	(void) flock(ds_fd, LOCK_UN);
	(void) close(ds_fd);

	/*
	 * Reset package variables
	 */
	ds_remove = FALSE;
	ds_fd = -1;
	ds_name[0] = '\0';
	ds_mode = 0;
	return DS_SUCCESS;
}

/*
 * ds_visit:
 *	Go into the dataset directory
 */
int
ds_visit()
{
	if (ds_fd < 0)
		return 0;
	return chdir(ds_name);
}

/*
 * ds_permission:
 *	Return the permission to the dataset based on client
 */
int
ds_permission(clp)
CLIENT	*clp;
{
#ifdef NGROUPS
	register int		i;
#endif

	if (ds_name[0] == '\0')
		return DS_PERM_RW;
	if (clp->uid == ds_uid)
		return (ds_mode >> 6) & 0x7;
#ifdef NGROUPS
	else {
		for (i = 0; i < clp->ngroups; i++)
			if (clp->groups[i] == ds_gid)
				return (ds_mode >> 3) & 0x7;
		return ds_mode & 0x7;
	}
#else
	else if (clp->gid == ds_gid)
		return (ds_mode >> 3) & 0x7;
	else
		return ds_mode & 0x7;
#endif
}

/*
 * ds_fchown:
 *	Set the ownership of the given file (descriptor)
 */
void
ds_fchown(fd)
int	fd;
{
	if (ds_fd < 0)
		return;
	(void) fchown(fd, ds_uid, ds_gid);
}

/*
 * ds_active:
 *	Is there a dataset opened?
 */
int
ds_active()
{
	return ds_fd;
}

/*
 * ds_set_times:
 *	Set access and modification times
 */
```

```
void
ds_set_times(atime, mtime)
time_t	atime, mtime;
{
	if (atime != 0)
		ds_atime = atime;
	if (mtime != 0)
		ds_mtime = mtime;
}

/*
 * ds_describe:
 *	Describe dataset to client
 */
void
ds_describe(fd)
int	fd;
{
	char	buf[BUFLEN];

	reply(fd, RS_ACCEPT, RT_DM, RG_OKAY, "Description follows");
	reply_begin_ASCII(fd);
	(void) sprintf(buf, "time %ld %ld", ds_mtime, ds_atime);
	reply_ASCII(fd, buf);
	reply_end_ASCII(fd);
}
```

dm/dataset.h

```
/*
 * $Header: dataset.h,v 1.3 88/08/08 09:46:28 conrad Exp $
 * $Log:        dataset.h,v $
 * Revision 1.3  88/08/08  09:46:28  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.2  88/07/29  14:59:53  conrad
 * Add functions for making proper account own dataset files
 *
 * Revision 1.1  88/07/25  13:15:21  conrad
 * Initial revision
 *
 */

#ifndef  DATASET_INCLUDE

#include "config.h"
#include <sys/file.h>

/*
 * Permission modes that dataset directories should have
 */
#define  DS_DEF_MODE    0775
#define  DS_LOCKFILE    ".locked_by"

/*
 * Permission modes on tables and files
 */
#define  DS_PERM_READ   R_OK
#define  DS_PERM_WRITE  W_OK
#define  DS_PERM_RW     (DS_PERM_READ | DS_PERM_WRITE)

/*
 * Maximum length of dataset name
 */
#define  DS_NAMESIZE    IO_SIZE

/*
 * Return values
 */
#define  DS_MORE        -6
#define  DS_WRONGFILE   -5
#define  DS_UNAVAIL     -4
#define  DS_EXISTS      -3
#define  DS_NOFILE      -2
#define  DS_NOPERM      -1

#define  DS_SUCCESS     0
#define  DS_READWRITE   0
#define  DS_READONLY    1

/*
 * List of access functions
 */
int      ds_open();             /* Open data set */
int      ds_create();           /* Create data set */
int      ds_delete();           /* Delete data set */
int      ds_cleanup();          /* Close data set */
int      ds_visit();            /* Go to data set directory */
int      ds_permission();       /* Return permission on dataset */
void     ds_fchown();           /* Change ownership to project */
int      ds_active();           /* Is there an open dataset? */
void     ds_set_times();        /* Set access and mod times */
void     ds_describe();         /* Describe dataset to client */

#define  DATASET_INCLUDE
#endif
```

dm/datatype.c

```
/*
 * $Header: datatype.c,v 1.6 88/10/04 15:04:26 conrad Exp $
 * $Log:		datatype.c,v $
 * Revision 1.6  88/10/04  15:04:26  conrad
 * Handle inputs as true streams with no record boundaries (newlines are
 * no longer expected as the last character of an input buffer)
 *
 * Revision 1.5  88/09/26  13:06:09  conrad
 * Check for EOF even in data-wait mode
 *
 * Revision 1.4  88/09/07  14:54:59  conrad
 * Use buffered writes instead of system call
 *
 * Revision 1.3  88/08/08  09:46:29  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.2  88/08/04  16:26:23  conrad
 * Support data type STRING and TABLE
 *
 * Revision 1.1  88/07/25  13:15:21  conrad
 * Initial revision
 *
 */

#include "datatype.h"
#include "error.h"
#include "reply.h"
#include "table.h"
#include "client.h"
#include <ctype.h>

#define	chartohex(c)	(isdigit(c)  ? (c) - '0' : (c) - 'A' + 10)
#define	hextochar(n)	((n) > 9 ? (n) - 10 + 'A' : (n) + '0')

static int		type_size[DM_S_NTYPE] = {
	1,		/* DM_S_INT1 */
	2,		/* DM_S_INT2 */
	4,		/* DM_S_INT4 */
	8,		/* DM_S_INT8 */
	4,		/* DM_S_FLOAT4 */
	8,		/* DM_S_FLOAT8 */
	16,		/* DM_S_FLOAT16 */
	1,		/* DM_S_CHAR */
	4,		/* DM_S_STRING */
	1,		/* DM_S_BYTE */
	4,		/* DM_S_TABLE */
};

/*
 * dt_size:
 *	Return the storage size of a data element
 */
int
dt_size(dtype, dsize)
int	dtype, dsize;
{
	if (type_size[dtype] == -1)
		return sizeof (int);
	else
		return type_size[dtype] * dsize;
}

/*
 * dt_supported:
 *	See if the given type is supported
 */
int
dt_supported(dtype)
int	dtype;
{
	switch (dtype) {
	 case DM_S_INT1:
	 case DM_S_INT2:
	 case DM_S_INT4:
	 case DM_S_FLOAT4:
	 case DM_S_FLOAT8:
	 case DM_S_CHAR:
	 case DM_S_BYTE:
	 case DM_S_STRING:
	 case DM_S_TABLE:
		return 0;
	 case DM_S_INT8:
	 case DM_S_FLOAT16:
	 default:
		return -1;
	}
}

/*
 * dt_send:
 *	Convert the given data type into printed form
 *	and send it on the file descriptor
 */
```

dm/datatype.c

```
void
dt_send(fd, dtype, length, data)
int	fd;
int	dtype;
int	length;
char	*data;
{
	void	send_bytes();
	void	send_shorts();
	void	send_longs();
	void	send_floats();
	void	send_doubles();

	switch (dtype) {
	 case DM_S_INT1:
	 case DM_S_BYTE:
	 case DM_S_CHAR:
		send_bytes(fd, data, length);
		break;
	 case DM_S_INT2:
		send_shorts(fd, (short *) data, length);
		break;
	 case DM_S_INT4:
		send_longs(fd, (long *) data, length);
		break;
	 case DM_S_FLOAT4:
		send_floats(fd, (float *) data, length);
		break;
	 case DM_S_FLOAT8:
		send_doubles(fd, (double *) data, length);
		break;
	 case DM_S_INT8:
	 case DM_S_FLOAT16:
	 case DM_S_STRING:
	 case DM_S_TABLE:
		error_internal("dt_convert unimplemented type");
		/* NOTREACHED */
	 default:
		error_internal("dt_convert unknown type");
		/* NOTREACHED */
	}
}

/*
 * send_bytes:
 *	Send stream of bytes to other end (each byte is converted to
 *	1 or 2 hex digits followed by a newline)
 */
static
void
send_bytes(fd, data, length)
int	fd;
char	*data;
int	length;
{
#define	CHUNK	(IO_SIZE / 3)			/* 3 = 2 hex bytes + newline */
	register int	i, n;
	register char	*bp;
	register char	*edata;
	char		buf[BUFLEN];

	edata = data + length;
	while (data < edata) {
		bp = buf;
		for (i = 0; i < CHUNK && data < edata; i++) {
			if ((n = (*data >> 4) & 0xf) != 0)
				*bp++ = hextochar(n);
			n = *data++ & 0xf;
			*bp++ = hextochar(n);
			*bp++ = '\n';
		}
		reply__write(fd, buf, bp - buf);
	}
#undef	CHUNK
}

/*
 * send_shorts:
 *	Send stream of shorts to other end (each short is converted to
 *	minimal number of hex bytes followed by a newline)
 */
static
void
send_shorts(fd, data, length)
int	fd;
short	*data;
int	length;
{
#define	CHUNK	(IO_SIZE / 5)			/* 5 = 4 hex bytes + newline */
	register int	i, j, n;
	register char	*bp;
	register short	*edata;
	char		buf[BUFLEN];
```

```
	edata = data + length;
	while (data < edata) {
		bp = buf;
		for (i = 0; i < CHUNK && data < edata; i++) {
			for (j = sizeof *data * BPB - 4; j > 0; j -= 4)
				if (((*data >> j) & 0xf) != 0)
					break;
			while (j >= 0) {
				n = (*data >> j) & 0xf;
				*bp++ = hextochar(n);
				j -= 4;
			}
			*bp++ = '\n';
			data++;
		}
		reply__write(fd, buf, bp - buf);
	}
#undef	CHUNK
}

/*
 * send_longs:
 *	Send stream of longs to other end (each longs is converted to
 *	minimal number of hex bytes followed by a newline)
 */
static
void
send_longs(fd, data, length)
int	fd;
long	*data;
int	length;
{
#define	CHUNK	(IO_SIZE / 9)			/* 9 = 8 hex bytes + newline */
	register int	i, j, n;
	register char	*bp;
	register long	*edata;
	char		buf[BUFLEN];

	edata = data + length;
	while (data < edata) {
		bp = buf;
		for (i = 0; i < CHUNK && data < edata; i++) {
			for (j = sizeof *data * BPB - 4; j > 0; j -= 4)
				if (((*data >> j) & 0xf) != 0)
					break;
			while (j >= 0) {
				n = (*data >> j) & 0xf;
				*bp++ = hextochar(n);
				j -= 4;
			}
			*bp++ = '\n';
			data++;
		}
		reply__write(fd, buf, bp - buf);
	}
#undef	CHUNK
}

/*
 * send_floats:
 *	Send a stream of floats to client
 */
static
void
send_floats(fd, data, length)
int	fd;
float	*data;
int	length;
{
#define	BSIZE	14
#define	CHUNK	(IO_SIZE / BSIZE)
	register char	*bp, *ebuf;
	register float	*edata;
	char		buf[BUFLEN];

	edata = data + length;
	ebuf = buf + sizeof buf;
	while (data < edata) {
		for (bp = buf; bp < ebuf && data < edata; bp += BSIZE)
			(void) sprintf(bp, "% .6e\n", *data++);
		reply__write(fd, buf, bp - buf);
	}
#undef	BSIZE
#undef	CHUNK
}

/*
 * send_doubles:
 *	Send a stream of doubles to client
 */
static
void
send_doubles(fd, data, length)
int	fd;
```

```c
double	*data;
int	length;
{
#define	BSIZE	24
#define	CHUNK	(IO_SIZE / BSIZE)
	register char		*bp, *ebuf;
	register double		*edata;
	char			buf[BUFLEN];

	edata = data + length;
	ebuf = buf + sizeof buf;
	while (data < edata) {
		for (bp = buf; bp < ebuf && data < edata; bp += BSIZE)
			(void) sprintf(bp, "% .16e\n", *data++);
		reply__write(fd, buf, bp - buf);
	}
#undef	BSIZE
#undef	CHUNK
}

/*
 * dt_str_out:
 *	Send strings to file descriptor
 */
void
dt_str_out(fd, len, data, ap)
int	fd;
int	len;
char	*data;
ATTR	*ap;
{
	register int		i;
	register long		*dp;
	register char		*cp;

	dp = (long *) data;
	for (i = 0; i < len; i++) {
		cp = STR_VALUE(ap, *dp++);
		reply__write(fd, cp, strlen(cp));
		reply__putchar(fd, '\n');
	}
}

/*
 * dt_input:
 *	Read printed data into a buffer and returns number of "bytes" consumed
 */
int
dt_input(clp, dtype, dsize, data, count, ap)
CLIENT	*clp;
int	dtype, dsize;
char	*data;
int	count;
ATTR	*ap;
{
	int	recv_bytes();
	int	recv_shorts();
	int	recv_longs();
	int	recv_floats();
	int	recv_doubles();
	int	n;

	switch (dtype) {
	 case DM_S_INT1:
	 case DM_S_CHAR:
	 case DM_S_BYTE:
		n = recv_bytes(clp, data, count / dsize);
		break;
	 case DM_S_INT2:
		n = recv_shorts(clp, (short *) data, count / dsize);
		break;
	 case DM_S_INT4:
		n = recv_longs(clp, (long *) data, count / dsize);
		break;
	 case DM_S_FLOAT4:
		n = recv_floats(clp, (float *) data, count / dsize);
		break;
	 case DM_S_FLOAT8:
		n = recv_doubles(clp, (double *) data, count / dsize);
		break;
	 case DM_S_STRING:
	 case DM_S_TABLE:
		n = recv_strings(clp, (long *) data, count / dsize, ap);
		break;
	 case DM_S_INT8:
	 case DM_S_FLOAT16:
		error_internal("dt_input failed");
		/* NOTREACHED */
	}
	if (n < 0)
		return -1;
	return n * dt_size(dtype, 1);
}
```

dm/datatype.c

```c
/*
 * recv_bytes:
 *	Receive some bytes from client and return number of
 *	bytes read
 */
static
int
recv_bytes(clp, buf, count)
CLIENT	*clp;
char	*buf;
int	count;
{
	register char	*cp, *mark;
	register int	n, nread;

	/*
	 * Get ASCII data into input buffer and init variables
	 */
	nread = 0;
	mark = CL_BUF(clp);
	*buf = 0;

	/*
	 * Look at the characters.  If the character is a newline, we
	 * know we've read a good value.  If not, then it must be data.
	 * We keep going (always marking the beginning of the current
	 * piece of datum) until we run out of data or satisfy the
	 * read.  In either case, we move the remaining data to the
	 * front of the input buffer.
	 */
	for (n = 0, cp = mark; n < CL_BUFCNT(clp) && count > 0; n++, cp++) {
		if (*cp == '\n') {
			nread++;
			buf++;
			mark = cp + 1;
			if (--count > 0)
				*buf = 0;
		}
		else
			*buf = (*buf << 4) | chartohex(*cp);
	}
	cl_consume(clp, mark - CL_BUF(clp));
	return nread;
}

/*
 * recv_shorts:
 *	Receive some shorts from client and return number of
 *	shorts read
 */
static
int
recv_shorts(clp, buf, count)
CLIENT	*clp;
short	*buf;
int	count;
{
	register char	*cp, *mark;
	register int	n, nread;

	nread = 0;
	mark = CL_BUF(clp);
	*buf = 0;
	for (n = 0, cp = mark; n < CL_BUFCNT(clp) && count > 0; n++, cp++) {
		if (*cp == '\n') {
			nread++;
			buf++;
			mark = cp + 1;
			if (--count > 0)
				*buf = 0;
		}
		else
			*buf = (*buf << 4) | chartohex(*cp);
	}
	cl_consume(clp, mark - CL_BUF(clp));
	return nread;
}

/*
 * recv_longs:
 *	Receive some longs from client and return number of
 *	longs read
 */
static
int
recv_longs(clp, buf, count)
CLIENT	*clp;
long	*buf;
int	count;
{
	register char	*cp, *mark;
	register int	n, nread;

	nread = 0;
```

```c
	mark = CL_BUF(clp);
	*buf = 0;
	for (n = 0, cp = mark; n < CL_BUFCNT(clp) && count > 0; n++, cp++) {
		if (*cp == '\n') {
			nread++;
			buf++;
			mark = cp + 1;
			if (--count > 0)
				*buf = 0;
		}
		else
			*buf = (*buf << 4) | chartohex(*cp);
	}
	cl_consume(clp, mark - CL_BUF(clp));
	return nread;
}

/*
 * recv_floats:
 *	Receive some floats from client and return number of
 *	floats read
 */
static
int
recv_floats(clp, buf, count)
CLIENT	*clp;
float	*buf;
int	count;
{
#define	BSIZE	14
	register char	*cp;
	register int	n, nread;
	double		atof();

	n = CL_BUFCNT(clp);
	cp = CL_BUF(clp);
	nread = 0;
	while (n >= BSIZE && count > 0) {
		*buf++ = atof(cp);
		nread++;
		count--;
		cp += BSIZE;
		n -= BSIZE;
	}
	cl_consume(clp, nread * BSIZE);
	return nread;
#undef	BSIZE
}

/*
 * recv_doubles:
 *	Receive some doubles from client and return number of
 *	doubles read
 */
static
int
recv_doubles(clp, buf, count)
CLIENT	*clp;
double	*buf;
int	count;
{
#define	BSIZE	24
	register char	*cp;
	register int	n, nread;
	double		atof();

	n = CL_BUFCNT(clp);
	cp = CL_BUF(clp);
	nread = 0;
	while (n >= BSIZE && count > 0) {
		*buf++ = atof(cp);
		nread++;
		count--;
		cp += BSIZE;
		n -= BSIZE;
	}
	cl_consume(clp, nread * BSIZE);
	return nread;
#undef	BSIZE
}

/*
 * recv_strings:
 *	Receive some strings from client and return number of
 *	strings read
 */
static
int
recv_strings(clp, buf, count, ap)
CLIENT	*clp;
long	*buf;
int	count;
ATTR	*ap;
{
```

dm/datatype.c

```
	register char	*cp, *mark;
	register int	n, nread;

	/*
	 * Get ASCII data into input buffer and init variables
	 */
	nread = 0;
	mark = CL_BUF(clp);

	/*
	 * Look at the characters.  If the character is a newline, we
	 * know we've read a good value.  If not, then it must be data.
	 * We keep going (always marking the beginning of the current
	 * piece of datum) until we run out of data or satisfy the
	 * read.  In either case, we move the remaining data to the
	 * front of the input buffer.
	 */
	for (n = 0, cp = mark; n < CL_BUFCNT(clp) && count > 0; n++, cp++)
		if (*cp == '\n') {
			nread++;
			count--;
			*cp = '\0';
			*buf++ = str_lookup(ap, mark);
			mark = cp + 1;
		}
	cl_consume(clp, mark - CL_BUF(clp));
	return nread;
}
```

dm/datatype.h

```
/*
 * $Header: datatype.h,v 1.2 88/08/04 16:26:26 conrad Exp $
 * $Log:          datatype.h,v $
 * Revision 1.2  88/08/04  16:26:26  conrad
 * Support data type STRING and TABLE
 *
 * Revision 1.1  88/07/25  13:15:22  conrad
 * Initial revision
 *
 */

#ifndef DATATYPE_INCLUDE

#include "config.h"
#include "dm_server.h"

/*
 * access functions
 */
int     dt_size();              /* Compute size of data element */
int     dt_supported();         /* Whether a data type is supported
void    dt_send();              /* Send data to client */
void    dt_str_out();           /* Send strings to client */
int     dt_input();             /* Read in data */
void    dt_str_in();            /* Read strings from client */

#define DATATYPE_INCLUDE
#endif
```

dm/dblock.c

```c
/*
 * $Header: dblock.c,v 1.10 89/03/02 22:01:09 pett Exp $
 * $Log:		dblock.c,v $
 * Revision 1.10  89/03/02  22:01:09  pett
 * declared emalloc/erealloc to return void *
 *
 * Revision 1.9  88/10/04  15:04:28  conrad
 * Handle inputs as true streams with no record boundaries (newlines are
 * no longer expected as the last character of an input buffer)
 *
 * Revision 1.8  88/09/26  13:05:14  conrad
 * Check for EOF even in data-wait mode
 *
 * Revision 1.7  88/09/22  15:49:46  conrad
 * Mark an attribute as modified if a cell is cleared
 *
 * Revision 1.6  88/09/07  14:54:40  conrad
 * Take out unnecessary include file
 *
 * Revision 1.5  88/08/04  18:49:44  conrad
 * Clear out data block as well as existence block
 *
 * Revision 1.4  88/08/04  16:26:27  conrad
 * Support data type STRING and TABLE
 *
 * Revision 1.3  88/07/29  15:00:38  conrad
 * Reverse sense of comparison for lock testing
 *
 * Revision 1.2  88/07/28  09:39:38  conrad
 * When adding or deleting rows, make sure all attributes are either
 * locked or available
 *
 * Revision 1.1  88/07/25  13:15:23  conrad
 * Initial revision
 *
 */

#include "table.h"
#include "lock.h"
#include "reply.h"
#include "error.h"
#include "datatype.h"

/*
 * dblock_alloc:
 *		Allocate a data block with given starting index and size
 */
DBLOCK *
dblock_alloc(start, size, esize, zero)
int		start, size;
int		esize, zero;
{
		register DBLOCK	*dp;
		extern void		*emalloc();

		dp = (DBLOCK *) emalloc(sizeof (DBLOCK));
		dp->start = start;
		dp->size = size;
		dp->next = NULL;
		dp->exist = (DEXIST) emalloc(size / BPB);
		dp->data = (void *) emalloc(esize * size);
		if (zero) {
				bzero((char *) dp->exist, size / BPB);
				bzero((char *) dp->data, esize * size);
		}
		return dp;
}

/*
 * dblock_release:
 *		Release a list of data blocks
 */
void
dblock_release(dp)
register DBLOCK	*dp;
{
		register DBLOCK	*next;

		while (dp != NULL) {
				next = dp->next;
				dblock_free(dp);
				dp = next;
		}
}

/*
 * dblock_free:
 *		Release data block
 */
void
dblock_free(dp)
register DBLOCK	*dp;
{
		(void) free((char *) dp->data);
```

```
		(void) free((char *) dp->exist);
		(void) free((char *) dp);
}

/*
 * dblock_add:
 *		Add a new row to an attribute
 */
void
dblock_add(ap, n)
ATTR	*ap;
int	n;
{
	register DBLOCK	*dp, *last;
	register int		start, esize;

	if (ap->ncell >= n)
		return;
	last = NULL;
	for (dp = ap->dblock; dp != NULL; dp = dp->next)
		last = dp;
	if (last != NULL && last->start + last->size > n)
		/* Already have space */
		return;

	/*
	 * We actually have to add the row.  If we didn't have any rows
	 * and the added row is numbered less than DE_FRAGMENT, then
	 * we allocate a small block.  If we have a fragment only, and
	 * the new row is beyond the fragment, we extend the fragment
	 * into a full block.  Otherwise, we allocate a full
	 * size block.
	 */
	esize = dt_size(ap->dtype, ap->dsize);
	if (last == NULL && n < DE_FRAGMENT)
		ap->dblock = dblock_alloc(0, DE_FRAGMENT, esize, TRUE);
	else {
		if (last != NULL && last->size == DE_FRAGMENT) {
			dp = dblock_alloc(0, DE_SIZE, esize, TRUE);
			(void) bcopy((char *) last->exist, (char *) dp->exist,
				last->size / BPB);
			(void) bcopy((char *) last->data, (char *) dp->data,
				esize * last->size);
			ap->dblock = dp;
			dblock_free(last);
			last = dp;
		}
		start = (last == NULL) ? 0 : last->start + last->size;
		while (dp->start + dp->size <= n) {
			dp = dblock_alloc(start, DE_SIZE, esize, TRUE);
			if (last == NULL)
				ap->dblock = dp;
			else
				last->next = dp;
			last = dp;
			start += DE_SIZE;
		}
	}
}

/*
 * dblock_find:
 *		Find the data block that contains a particular row
 */
DBLOCK *
dblock_find(dp, inum)
register DBLOCK	*dp;
register int		inum;
{
	register int		n;

	while (dp != NULL) {
		n = inum - dp->start;
		if (n >= 0 && n < dp->size)
			break;
		dp = dp->next;
	}
	return dp;
}

/*
 * inst_new:
 *		Create a new instance for a table
 */
int
inst_new(clp, tname, inum)
CLIENT	*clp;
char	*tname;
int	*inum;
{
	register TABLE	*tp;
	register ATTR	*ap;
	register int	max;
	extern time_t	time();
```

```
	/*
	 * Find the table
	 */
	tp = tbl_find(tname);
	if (tp == NULL || tp->deleted)
		return TBL_NOSUCH;
	if (lock_mode(tp->lock, clp) != LOCK_WRITE)
		return TBL_NOPERM;

	/*
	 * Make sure all attributes are either write-locked by client
	 * or not locked at all
	 */
	for (ap = tp->attr; ap != NULL; ap = ap->next)
		if (lock_by_other(ap->lock, clp) < 0)
			return ATTR_NOPERM;

	/*
	 * Add the new instance.  We don't actually bump
	 * the instance count since no data has gone in yet.
	 */
	max = inst_max(tp) + 1;
	for (ap = tp->attr; ap != NULL; ap = ap->next)
		dblock_add(ap, max);
	*inum = max;
	tp->mtime = time((time_t *) 0);
	return INST_SUCCESS;
}

/*
 * inst_delete:
 *	Delete an instance
 */
int
inst_delete(clp, tname, inum)
CLIENT	*clp;
char	*tname;
int	inum;
{
	register TABLE	*tp;
	register ATTR	*ap;
	void		cell_clear();

	/*
	 * Find the table
	 */
	tp = tbl_find(tname);
	if (tp == NULL || tp->deleted)
		return TBL_NOSUCH;
	if (lock_mode(tp->lock, clp) != LOCK_WRITE)
		return TBL_NOPERM;

	/*
	 * Make sure all attributes are either write-locked by client
	 * or not locked at all
	 */
	for (ap = tp->attr; ap != NULL; ap = ap->next)
		if (lock_by_other(ap->lock, clp) < 0)
			return ATTR_NOPERM;

	/*
	 * Empty out the cells for all the attributes
	 */
	for (ap = tp->attr; ap != NULL; ap = ap->next)
		(void) cell_clear(ap, inum, inum);

	tp->mtime = time((time_t *) 0);
	return INST_SUCCESS;
}

/*
 * cell_clear:
 *	Clear out cells of an attribute
 */
static
void
cell_clear(ap, from, to)
ATTR	*ap;
int	from, to;
{
	register DBLOCK	*dp;
	register int	i, n;
	register int	byte, bit;
	int		before, after;
	extern time_t	time();

	/*
	 * Find the beginning of the stuff to be cleared
	 */
	if (from > ap->ncell)
		return;
	if (++to > ap->ncell + 1)
		to = ap->ncell + 1;
```

dm/dblock.c

```
	dp = dblock_find(ap->dblock, from);

	/*
	 * Zap out the data from beginning to end
	 */
	for (i = from; i < to; i++) {
		n = i - dp->start;
		if (n == dp->size) {
			dp = dp->next;
			n = i - dp->start;
		}
		byte = n / BPB;
		bit = 1 << (n % BPB);
		if ((dp->exist[byte] & bit) != 0) {
			dp->exist[byte] &= ~bit;
			ap->ndata--;
		}
	}
	ap->mtime = time((time_t *) 0);
	ap->modified = TRUE;

	/*
	 * Now update the ncell field of the attribute
	 */
	if (to < ap->ncell)
		return;
	before = inst_max(ap->table);
	i = ap->ncell;
	while (i >= 0) {
		dp = dblock_find(ap->dblock, i);
		for (n = dp->size - 1; n >= 0; n--) {
			byte = n / BPB;
			bit = 1 << (n % BPB);
			if ((dp->exist[byte] & bit) != 0) {
				ap->ncell = dp->start + n;
				goto found;
			}
		}
		i = dp->start - 1;
	}
	ap->ncell = -1;
found:
	after = inst_max(ap->table);
	if (before != after)
		ap->table->mtime = ap->mtime;
	return;
}

/*
 * cell_empty:
 *	Empty out a cell
 */
int
cell_empty(clp, tname, aname, from, to)
CLIENT	*clp;
char	*tname, *aname;
int	from, to;
{
	register TABLE	*tp;
	register ATTR	*ap;
	void		cell_clear();

	/*
	 * Make sure that the table and attribute exist and locked properly
	 */
	tp = tbl_find(tname);
	if (tp == NULL || tp->deleted)
		return TBL_NOSUCH;
	for (ap = tp->attr; ap != NULL; ap = ap->next)
		if (strcmp(ap->name, aname) == 0)
			break;
	if (ap == NULL)
		return ATTR_NOSUCH;
	if (lock_mode(ap->lock, clp) != LOCK_WRITE)
		return ATTR_NOPERM;

	cell_clear(ap, from, to);
	return INST_SUCCESS;
}

/*
 * inst_max:
 *	Return the row number of the last instance in a table with
 *	any data associated with it.
 */
int
inst_max(tp)
TABLE	*tp;
{
	register ATTR	*ap;
	register int	n;

	n = -1;
	for (ap = tp->attr; ap != NULL; ap = ap->next)
```

```
		if (ap->ncell > n)
			n = ap->ncell;
	return n;
}

/*
 * cell_read:
 *	Read a range of cells
 */
int
cell_read(clp, tname, aname, from, to)
CLIENT	*clp;
char	*tname, *aname;
int	from, to;
{
	register TABLE	*tp;
	register ATTR	*ap;
	register DBLOCK	*dp, *startdp;
	register int	start, i, n;
	register int	byte, bit;
	void		send_data();

	/*
	 * Make sure table and attribute exist
	 * and attribute is locked
	 */
	tp = tbl_find(tname);
	if (tp == NULL || tp->deleted)
		return TBL_NOSUCH;
	for (ap = tp->attr; ap != NULL; ap = ap->next)
		if (strcmp(ap->name, aname) == 0)
			break;
	if (ap == NULL)
		return ATTR_NOSUCH;
	if (lock_mode(ap->lock, clp) == LOCK_NONE)
		return ATTR_NOPERM;

	/*
	 * Normalize the from and to indices
	 */
	if (from > ap->ncell)
		return CELL_SUCCESS;
	if (to > ap->ncell)
		to = ap->ncell;

	/*
	 * Loop through the cells and figure out ranges
	 */
	start = -1;
	dp = dblock_find(ap->dblock, from);
	for (i = from; i <= to; i++) {
		if (i >= dp->start + dp->size)
			dp = dp->next;
		n = i - dp->start;
		byte = n / BPB;
		bit = 1 << (n % BPB);
		if ((dp->exist[byte] & bit) != 0) {
			if (start == -1) {
				start = i;
				startdp = dp;
			}
		}
		else {
			if (start != -1) {
				send_data(clp, ap, start, i - 1, startdp);
				start = -1;
			}
		}
	}
	if (start != -1)
		send_data(clp, ap, start, to, startdp);
	return CELL_SUCCESS;
}

/*
 * send_data:
 *	Send a range of data to client
 */
static
void
send_data(clp, ap, from, to, dp)
CLIENT	*clp;
ATTR	*ap;
int	from, to;
DBLOCK	*dp;
{
	register int	i, esize;
	register int	start, len, remain;

	esize = dt_size(ap->dtype, ap->dsize);
	i = from;
	while (i <= to) {
		/*
		 * Send data from this data block
```

```
		 */
		start = i - dp->start;
		len = dp->size - start;
		remain = to - i + 1;
		if (len > remain)
			len = remain;
		reply_data(CL_FILENO(clp), CL_DATAMODE(clp), ap->dtype,
			ap->dsize, i, len,
			((char *) dp->data) + start * esize, ap);

		/*
		 * Advance to next block
		 */
		i += len;
		dp = dp->next;
	}
}

/*
 * cell_write:
 *	Handle a write command from user.
 */
int
cell_write(clp, tname, aname, from, to)
CLIENT	*clp;
char	*tname, *aname;
int	from, to;
{
	register TABLE	*tp;
	register ATTR	*ap;
	register DBLOCK	*dp, *ndp;
	register int	n, esize, i;
	struct iovec	*iov, *ip;
	extern void	*emalloc();

	/*
	 * Make sure table and attribute exist
	 * and attribute is locked
	 */
	tp = tbl_find(tname);
	if (tp == NULL || tp->deleted)
		return TBL_NOSUCH;
	for (ap = tp->attr; ap != NULL; ap = ap->next)
		if (strcmp(ap->name, aname) == 0)
			break;
	if (ap == NULL)
		return ATTR_NOSUCH;
	if (lock_mode(ap->lock, clp) != LOCK_WRITE)
		return ATTR_NOPERM;

	/*
	 * We don't mark the data as present until all the data
	 * has come over because the client may die after sending
	 * the WRITE command but before sending the data.  We
	 * will mark the data as present AFTER all the data have
	 * arrived.  So we stick the appropriate information (including
	 * an iovec structure which contains pointers to data storage
	 * areas) in the client state and say things are okay.
	 */
	dblock_add(ap, to);			/* Make sure there is space */
	n = 1;					/* Count # iovec's needed */
	dp = dblock_find(ap->dblock, from);
	for (ndp = dp; ndp->start + ndp->size <= to; ndp = ndp->next)
		n++;
	esize = dt_size(ap->dtype, ap->dsize);

	/*
	 * We fill in an iovec structure by filling in the first
	 * block by hand, filling in intermediate blocks in a loop,
	 * and the last block by hand again
	 */
	iov = (struct iovec *) emalloc(n * sizeof (struct iovec));
	iov[0].iov_base = ((char *) dp->data) + ((from - dp->start) * esize);
	i = (dp->start + dp->size < to + 1) ? (dp->start + dp->size) : to + 1;
	iov[0].iov_len = (i - from) * esize;
	dp = dp->next;
	for (ip = iov + 1, i = 2; i < n; ip++, i++) {
		ip->iov_base = (char *) dp->data;
		ip->iov_len = dp->size * esize;
		dp = dp->next;
	}
	if (i == n) {			/* Only if there is more than one block */
		ip->iov_base = (char *) dp->data;
		ip->iov_len = (to - dp->start + 1) * esize;
	}

	cl_data_wait(clp, ap->dtype, ap->dsize, n, iov, (void *) ap, from, to);
	return CELL_SUCCESS;
}

/*
 * cell_data:
 *	Read data into cells
 */
```

dm/dblock.c

```
int
cell_data(clp)
CLIENT  *clp;
{
        register int        i, n;
        register int        byte, bit;
        register ATTR       *ap;
        register DBLOCK     *dp;
        int                 from, to;
        struct iovec        *ip;

        for (i = 0, ip = CL_IOV(clp); i < CL_IOVCNT(clp); i++, ip++)
                if (ip->iov_len > 0)
                        break;
        if (i >= CL_IOVCNT(clp))
                error_internal("cell_data failed");

        ap = (ATTR *) CL_ATTRIBUTE(clp);
        if (CL_DATAMODE(clp) == REPLY_IN_BINARY
        &&  CL_DTYPE(clp) != DM_S_STRING && CL_DTYPE(clp) != DM_S_TABL
                while (i < CL_IOVCNT(clp)) {
                        if (CL_BUFCNT(clp) >= ip->iov_len) {
                                (void) bcopy(CL_BUF(clp),
                                        (char *) ip->iov_base, ip->iov_len);
                                cl_consume(clp, ip->iov_len);
                                ip->iov_len = 0;
                                ip++;
                                i++;
                        }
                        else {
                                (void) bcopy(CL_BUF(clp),
                                        (char *) ip->iov_base, CL_BUFCNT
                                ip->iov_base += CL_BUFCNT(clp);
                                ip->iov_len -= CL_BUFCNT(clp);
                                cl_consume(clp, CL_BUFCNT(clp));
                                break;
                        }
                }
        }
        else {
                while (i < CL_IOVCNT(clp)) {
                        n = dt_input(clp, CL_DTYPE(clp), CL_DSIZE(clp),
                                ip->iov_base, ip->iov_len, ap);
                        ip->iov_len -= n;
                        ip->iov_base += n;
                        if (ip->iov_len <= 0) {
                                ip++;
                                i++;
                        }
                }
        }
        if (i < CL_IOVCNT(clp))
                return -1;

        /*
         * We've read in all the data we expected.  Now we can set
         * all the data as present.
         */
        from = CL_FROM(clp);
        to = CL_TO(clp);
        dp = dblock_find(ap->dblock, from);
        i = (dp->start + dp->size - 1 < to) ? (dp->start + dp->size - 1) : to;
        while (i >= from) {
                n = i-- - dp->start;
                byte = n / BPB;
                bit = 1 << (n % BPB);
                if ((dp->exist[byte] & bit) == 0) {
                        dp->exist[byte] |= bit;
                        ap->ndata++;
                }
        }
        dp = dp->next;
        for (i = 2; i < CL_IOVCNT(clp); i++) {
                for (n = 0; n < dp->size; n++) {
                        byte = n / BPB;
                        bit = 1 << (n % BPB);
                        if ((dp->exist[byte] & bit) == 0) {
                                dp->exist[byte] |= bit;
                                ap->ndata++;
                        }
                }
                dp = dp->next;
        }
        if (i == CL_IOVCNT(clp)) {
                for (i = dp->start; i < to; i++) {
                        byte = n / BPB;
                        bit = 1 << (n % BPB);
                        if ((dp->exist[byte] & bit) == 0) {
                                dp->exist[byte] |= bit;
                                ap->ndata++;
                        }
                }
        }
        if (to > ap->ncell)
```

```
			ap->ncell = to;
		ap->mtime = time((time_t *) 0);
		ap->modified = TRUE;
		return 0;
}
```

dm/dfio.c

```c
/*
 * $Header: dfio.c,v 1.9 89/03/02 21:43:09 pett Exp $
 * $Log:	dfio.c,v $
 * Revision 1.9  89/03/02  21:43:09  pett
 * changed emalloc declaration to void *
 *
 * Revision 1.8  89/02/23  18:24:49  pett
 * changed some declarations to take advantage of machdep.h
 *
 * Revision 1.7  88/09/15  23:04:15  conrad
 * Make sure input and output agree on what is in a file
 *
 * Revision 1.6  88/09/14  14:29:01  conrad
 * Change variable "table" to "table_list" so dbx doesn't choke
 *
 * Revision 1.5  88/09/09  16:51:02  conrad
 * Check filename against filename, not table name
 *
 * Revision 1.4  88/08/08  09:46:31  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.3  88/08/04  16:26:29  conrad
 * Support data type STRING and TABLE
 *
 * Revision 1.2  88/07/29  15:00:57  conrad
 * Make sure proper account owns dataset files
 *
 * Revision 1.1  88/07/25  13:15:23  conrad
 * Initial revision
 *
 */

#include "table.h"
#include "datatype.h"
#include "dataset.h"
#include <pwd.h>

/*
 * This file implements the I/O operations to and from data files within
 * a data set.  The format of the file is:
 *
 *	Header:	*	sizeof HIO = 28 bytes
 *	Tables:	MAGIC			2 byte magic number
 *		Table structure		sizeof TIO
 *		Table name		variable length (has null)
 *		Attribute structure	sizeof AIO
 *		Attribute name		variable length (has null)
 *		Data blocks		bitfield + data
 */

#define	MAGIC	((short) 0x4d57)
#define	VERSION	1
#define	HMAGIC	"MMWB datafile version %d\n"

typedef struct hio	{		/* Header with version number */
	char	header[28];
}	HIO;

typedef struct tio	{		/* Table structure */
	int	class;
	int	nattr;
	UIDTYPE	uid;
	GIDTYPE	gid;
	time_t	mtime, atime;
	int	namelen;
}	TIO;

typedef struct aio	{		/* Attribute structure */
	short	dtype, dsize;
	int	ncell;
	int	ndata;
	UIDTYPE	uid;
	GIDTYPE	gid;
	time_t	mtime, atime;
	int	namelen;
}	AIO;

/*
 * tbl_read:
 *	Read a table in from file
 */
int
tbl_read(fp, filename)
FILE	*fp;
char	*filename;
{
	register int	i;
	HIO		h;
	TIO		t;
	TABLE		*tp;
	int		version;
	short		magic;
	char		*name;
```

```
	struct passwd	*pp;
	int		get_attribute();
	char		*copy_string();
	extern void	*emalloc();

	/*
	 * Check the header
	 */
	if (fread((char *) &h, sizeof h, 1, fp) != 1)
		return 0;
	if (sscanf(h.header, HMAGIC, &version) != 1)
		return 0;
	if (version != VERSION)
		return -1;

	/*
	 * Read in the tables
	 */
	while (fread((char *) &magic, sizeof magic, 1, fp) == 1) {
		/* Check magic number for consistency */
		if (magic != MAGIC)
			return -1;

		/* Read table header & name */
		if (fread((char *) &t, sizeof t, 1, fp) != 1)
			return -1;
		name = (char *) emalloc(t.namelen);
		if (fread(name, 1, t.namelen, fp) != t.namelen) {
			(void) free(name);
			return -1;
		}

		/* Create the table */
		tp = (TABLE *) emalloc(sizeof (TABLE));
		tp->next = table_list;
		table_list = tp;
		tp->name = name;
		tp->filename = copy_string(filename);
		tp->class = t.class;
		tp->attr = NULL;
		tp->lock = lock_find(LOCK_TABLE, (void *) tp);
		tp->uid = t.uid;
		tp->gid = t.gid;
		pp = getpwuid((int) tp->uid);
		if (pp == NULL)
			tp->owner = "nobody";
		else
			tp->owner = copy_string(pp->pw_name);
		tp->deleted = FALSE;
		tp->modified = FALSE;
		tp->holder = NULL;
		tp->mtime = t.mtime;
		tp->atime = t.atime;
		tp->nattr = t.nattr;

		/* Read attributes */
		for (i = 0; i < tp->nattr; i++)
			if (get_attribute(fp, tp) < 0) {
				attr_release(tp->attr);
				return -1;
			}
	}

	return 0;
}

/*
 * get_attribute:
 *	Read an attribute from a file
 */
static
int
get_attribute(fp, tp)
FILE	*fp;
TABLE	*tp;
{
	AIO		a;
	ATTR		*ap;
	char		*name;
	struct passwd	*pp;
	int		get_dblock();
	extern void	*emalloc();

	/*
	 * Read in attribute header and name
	 */
	if (fread((char *) &a , sizeof a, 1, fp) != 1)
		return -1;
	name = (char *) emalloc(a.namelen);
	if (fread(name, 1, a.namelen, fp) != a.namelen) {
		(void) free(name);
		return -1;
	}
```

```
	/*
	 * Create attribute
	 */
	ap = (ATTR *) emalloc(sizeof (ATTR));
	ap->next = tp->attr;
	tp->attr = ap;
	ap->name = name;
	ap->dsize = a.dsize;
	ap->dtype = a.dtype;
	ap->ncell = a.ncell;
	ap->ndata = a.ndata;
	ap->mtime = a.mtime;
	ap->atime = a.atime;
	ap->uid = a.uid;
	ap->gid = a.gid;
	pp = getpwuid((int) ap->uid);
	if (pp == NULL)
		ap->owner = "nobody";
	else
		ap->owner = copy_string(pp->pw_name);
	ap->lock = lock_find(LOCK_ATTR, (void *) ap);
	ap->table = tp;
	ap->modified = FALSE;
	ap->dblock = NULL;

	/*
	 * Read in data blocks
	 */
	if (ap->ncell >= 0) {
		if (get_dblock(fp, ap) < 0) {
			attr_free(ap);
			return -1;
		}
		if (ap->dtype == DM_S_STRING || ap->dtype == DM_S_TABLE)
			if (str_read(fp, ap) < 0)
				return -1;
	}
	else if (ap->dtype == DM_S_STRING || ap->dtype == DM_S_TABLE)
		str_init(ap);

	return 0;
}

/*
 * get_dblock:
 *	Read in data blocks
 */

static
int
get_dblock(fp, ap)
FILE	*fp;
ATTR	*ap;
{
	register int	size, count;
	DBLOCK		*dp, *last;
	int		esize, bsize, start;
	extern void	*emalloc();

	start = 0;
	bsize = DE_FRAGMENT;
	esize = dt_size(ap->dtype, ap->dsize);
	for (count = ap->ncell; count >= 0; count -= DE_SIZE) {
		size = (count > DE_SIZE) ? DE_SIZE : count;
		if (size > bsize)
			bsize = DE_SIZE;
		dp = dblock_alloc(start, bsize, esize, FALSE);
		if (ap->dblock == NULL)
			last = ap->dblock = dp;
		else {
			last->next = dp;
			last = dp;
		}
		if (fread((char *) dp->exist, bsize / BPB, 1, fp) != 1) {
			attr_free(ap);
			return -1;
		}
		if (fread((char *) dp->data, esize, bsize, fp) != bsize) {
			attr_free(ap);
			return -1;
		}
		start += size;
	}

	return 0;
}

/*
 * Here begins the output routines.  The strategy is to construct
 * a list of structure, one for each file.  The structure also contains
 * the list of tables that belong in that file.  Using the list of
 * structures, we can determine which files need to be rewritten.
 */

typedef struct tlist {
```

dm/dfio.c

```
	struct tlist		*next;
	TABLE			*table;
}	TLIST;

typedef struct flist {
	struct flist		*next;
	char			*filename;
	TLIST			*tlist;
}	FLIST;

/*
 * tbl_flush:
 *	Flush all tables out to disk
 */
void
tbl_flush()
{
	register FLIST		*fp, *nfp;
	register TLIST		*tlp, *ntlp;
	register FLIST		*flist;
	register TABLE		*tp;
	register ATTR		*ap;
	int			flush, ntable;
	void			write_tables();
	extern void		*emalloc();

	if (ds_active() < 0)
		return;

	/*
	 * Construct the list of structures
	 */
	flist = NULL;
	for (tp = table_list; tp != NULL; tp = tp->next) {
		tlp = (TLIST *) emalloc(sizeof (TLIST));
		tlp->table = tp;
		for (fp = flist; fp != NULL; fp = fp->next)
			if (strcmp(fp->filename, tp->filename) == 0)
				break;
		if (fp != NULL)
			tlp->next = fp->tlist;
		else {
			tlp->next = NULL;
			fp = (FLIST *) emalloc(sizeof (FLIST));
			fp->next = flist;
			fp->filename = tp->filename;
			flist = fp;
		}
		fp->tlist = tlp;
	}

	/*
	 * Loop through the list of structures to see which ones
	 * need to be flushed out
	 */
	for (fp = flist; fp != NULL; fp = fp->next) {
		flush = FALSE;
		ntable = 0;
		for (tlp = fp->tlist; tlp != NULL; tlp = tlp->next) {
			tp = tlp->table;
			if (!tp->deleted)
				ntable++;
			if (flush)
				continue;
			/*
			 * If the table structure changed or any data
			 * blocks changed, we must flush table to disk
			 */
			if (tp->modified)
				flush = TRUE;
			else
				for (ap = tp->attr; ap != NULL; ap = ap->next)
					if (ap->modified) {
						flush = TRUE;
						break;
					}
		}
		if (ntable == 0)
			(void) unlink(fp->filename);
		else if (flush)
			write_tables(fp);

		/*
		 * Release all the allocated space
		 */
		for (tlp = fp->tlist; tlp != NULL; tlp = ntlp) {
			ntlp = tlp->next;
			(void) free((char *) tlp);
		}
	}
#ifdef SAFE
	/*
	 * If all writes are success, move all files to the right places
	 */
```

```
		move_tables();
#endif
		for (fp = flist; fp != NULL; fp = nfp) {
			nfp = fp->next;
			(void) free((char *) fp);
		}
}

/*
 * write_tables:
 *		Write a set of tables out to a file
 */
static
void
write_tables(flp)
FLIST	*flp;
{
		register TLIST		*tlp;
		register FILE		*fp;
		HIO			h;
		void			put_table();

#ifdef SAFE
		/*
		 * Write to another file for now.  If all is successful,
		 * files will be moved into the proper places
		 */
#else
		if ((fp = fopen(flp->filename, "w")) == NULL) {
			syslog(LOG_ERR, "%s: %m", flp->filename);
			return;
		}
		ds_fchown(fileno(fp));
#endif
		(void) sprintf(h.header, HMAGIC, VERSION);
		(void) fwrite((char *) &h, sizeof h, 1, fp);
		for (tlp = flp->tlist; tlp != NULL; tlp = tlp->next)
			put_table(fp, tlp->table);
		(void) fclose(fp);
}

/*
 * put_table:
 *		Write out a table to a file stream
 */
static
void
put_table(fp, tp)
FILE	*fp;
TABLE	*tp;
{
		TIO	t;
		ATTR	*ap;
		short	magic;
		void	put_attribute();

		magic = MAGIC;
		(void) fwrite((char *) &magic, sizeof magic, 1, fp);
		t.class = tp->class;
		t.nattr = tp->nattr;
		t.uid = tp->uid;
		t.gid = tp->gid;
		t.mtime = tp->mtime;
		t.atime = tp->atime;;
		t.namelen = strlen(tp->name) + 1;
		(void) fwrite((char *) &t, sizeof t, 1, fp);
		(void) fwrite(tp->name, 1, t.namelen, fp);
		for (ap = tp->attr; ap != NULL; ap = ap->next)
			put_attribute(fp, ap);
}

/*
 * put_attribute:
 *		Write an attribute out to a file stream
 */
static
void
put_attribute(fp, ap)
FILE	*fp;
ATTR	*ap;
{
		AIO	a;
		void	put_dblock();

		/*
		 * Write attribute header
		 */
		a.dtype = ap->dtype;
		a.dsize = ap->dsize;
		a.ncell = ap->ncell;
		a.ndata = ap->ndata;
		a.uid = ap->uid;
		a.gid = ap->gid;
		a.mtime = ap->mtime;
```

```
	a.atime = ap->atime;
	a.namelen = strlen(ap->name) + 1;
	(void) fwrite((char *) &a, sizeof a, 1, fp);
	(void) fwrite(ap->name, 1, a.namelen, fp);

	/*
	 * Write individual data blocks
	 */
	if (ap->ncell >= 0) {
		put_dblock(fp, ap);
		if (ap->dtype == DM_S_STRING || ap->dtype == DM_S_TABLE)
			str_write(fp, ap);
	}
}

/*
 * put_dblock:
 *	Write data blocks out to file stream
 */
static
void
put_dblock(fp, ap)
FILE	*fp;
ATTR	*ap;
{
	register DBLOCK	*dp;
	register int		count, size, esize;

	size = (ap->ncell >= DE_FRAGMENT) ? DE_SIZE : DE_FRAGMENT;
	esize = dt_size(ap->dtype, ap->dsize);
	for (count = ap->ncell, dp = ap->dblock; count >= 0;
	count -= size, dp = dp->next) {
		(void) fwrite((char *) dp->exist, size / BPB, 1, fp);
		(void) fwrite((char *) dp->data, esize, size, fp);
	}
}
```

dm/dm.c

```
/*
 * $Header: dm.c,v 1.8 89/02/23 18:02:52 pett Exp $
 * $Log:        dm.c,v $
 * Revision 1.8  89/02/23  18:02:52  pett
 * changed some declarations to take advantage of machdep.h
 *
 * Revision 1.7  88/10/17  11:48:15  conrad
 * Don't delay writes for TCP connections
 *
 * Revision 1.6  88/10/06  12:45:38  conrad
 * Add statistics gathering
 *
 * Revision 1.5  88/09/14  14:29:09  conrad
 * Stop recording times in syslog
 *
 * Revision 1.4  88/09/07  14:51:43  conrad
 * Put in code for measuring idle time
 *
 * Revision 1.3  88/08/04  18:36:21  conrad
 * Dump core on appropriate signals
 *
 * Revision 1.2  88/08/04  16:25:44  conrad
 * Don't play with stop and continue signals
 *
 * Revision 1.1  88/07/25  13:15:24  conrad
 * Initial revision
 *
 */

#include "config.h"
#include "dm_server.h"
#include "client.h"
#include "reply.h"
#include "error.h"
#include "dataset.h"
#include <sys/time.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <signal.h>

#ifdef TIMES
#define diff_time(l,e) \
        (((l)->tv_sec - (e)->tv_sec) * 1000000 + ((l)->tv_usec - (e)->tv_usec))
#endif

#ifdef STATS
int     nlock_com, ntotal_com;
#endif

static char         *usage_msg[] = {
        "dm -C cid [-d]",
        "-C\tconversation identifier",
        NULL,
};

static char         *cid = "project@host.domain";

/*
 * dm:
 *      Data manager.  This process uses a non-preemptive multi-tasking
 *      approach to handle multiple clients.  Each client is guaranteed
 *      to return to the main loop when blocking (for I/O, locks, etc).
 */
main(ac, av)
int     ac;
char    **av;
{
        register int        c;
        fd_set              read_fds;
        int                 fd;
        struct sockaddr     sa;
        int                 salen;
        int                 debug;
#ifdef TIMES
        struct timeval      before, after;
        long                busy, idle;
#endif
        extern int          optind;
        extern char         *optarg;
        void                make_socket();
        int                 check_socket();
        int                 bomb();
#ifdef vax
        extern int          _pw_stayopen;

        _pw_stayopen = TRUE;                    /* BSD hack */
#endif
        while ((c = getopt(ac, av, "C:")) != EOF)
                switch (c) {
                 case 'C':
                        cid_set(optarg);
                        break;
                 default:
```

```
				usage(usage_msg);
				/* NOTREACHED */
			}

	debug = isatty(0);
	if (debug)
			make_socket();
	else
#ifdef LOG_LOCAL4
			openlog("mmwb-dm", LOG_PID, LOG_LOCAL4);
#else
			openlog("mmwb-dm", LOG_PID);
#endif

	for (c = 1; c < NSIG; c++)
			(void) signal(c, bomb);
	(void) signal(SIGPIPE, SIG_IGN);
	(void) signal(SIGTERM, SIG_IGN);		/* ignore dispatcher kills */
	(void) signal(SIGTSTP, SIG_DFL);
	(void) signal(SIGCONT, SIG_DFL);

	/*
	 * Make sure we're really waiting for clients
	 */
	if (check_socket() < 0) {
			syslog(LOG_ERR, "check_socket failed");
			exit(1);
	}

more:
	/*
	 * Now wait for clients.  We use a do-while loop to make sure
	 * that we accept at least one client first.
	 */
#ifdef TIMES
	(void) gettimeofday(&after, (struct timezone *) NULL);
	busy = idle = 0;
#endif
	do {
			FD_ZERO(&read_fds);
			FD_SET(0, &read_fds);
			cl_fdmask(&read_fds);
#ifdef TIMES
			(void) gettimeofday(&before, (struct timezone *) NULL);
			busy += diff_time(&before, &after);
#endif
			if (select(FD_SETSIZE, &read_fds, (fd_set *) NULL,
				(fd_set *) NULL, (struct timeval *) NULL) == -1) {
					syslog(LOG_ERR, "select: %m");
					error_internal("select() error");
					/* NOTREACHED */
			}
#ifdef TIMES
			(void) gettimeofday(&after, (struct timezone *) NULL);
			idle += diff_time(&after, &before);
#endif
			cl_input(&read_fds);
			if (FD_ISSET(0, &read_fds)) {
					/* New client connection request */
					salen = sizeof sa;
					if ((fd = accept(0, &sa, &salen)) < 0)
							syslog(LOG_ERR, "accept: %m");
					else {
#ifdef TCP_NODELAY
							int		on = 1;

							if (setsockopt(fd, IPPROTO_TCP, TCP_NODELAY,
							&on, sizeof on) < 0)
									syslog(LOG_ERR, "setsockopt: %m");
#endif
							(void) cl_new(fd);
							reply(fd, RS_ACCEPT, RT_OTHER, 0, cid);
							reply_end(fd);
					}
			}
	} while (clist != NULL);

	if (ds_cleanup() == DS_MORE)
			goto more;
#ifdef TIMES
	(void) gettimeofday(&before, (struct timezone *) NULL);
	busy += diff_time(&before, &after);
	syslog(LOG_INFO, "busy=%d, idle=%d", busy, idle);
#endif
#ifdef STATS
	syslog(LOG_INFO, "%d locks/%d commands", nlock_com, ntotal_com);
#endif
	exit(0);
}

/*
 * bomb:
 *	Log a message about bombing out
 */
```

dm/dm.c

```
static
SIGNALTYPE
bomb(sig)
int         sig;
{
        switch (sig) {
         case SIGQUIT:
         case SIGILL:
         case SIGTRAP:
         case SIGIOT:
         case SIGEMT:
         case SIGFPE:
         case SIGBUS:
         case SIGSEGV:
         case SIGSYS:
                syslog(LOG_ERR, "signal %d, core dumped", sig);
                abort();
         default:
                syslog(LOG_ERR, "signal %d", sig);
                exit(1);
        }
}

/*
 * cid_set:
 *      Set the conversation ID
 */
cid_set(id)
char        *id;
{
        cid = id;
}

/*
 * cid_verify:
 *      Make sure id and conversation ID match
 */
cid_verify(id)
char        *id;
{
        if (strcmp(id, cid) != 0)
                return -1;
        return 0;
}

/*
 * make_socket:
 *      Create a socket and bind it to an Internet port
 *      on file descriptor 0.  ONLY CALLED IN DEBUG MODE.
 */
static
void
make_socket()
{
        register int            fd;
        struct sockaddr_in      sa;
        int                     salen;
#ifndef ntohs
        unsigned short          ntohs();
#endif

        if ((fd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
                syslog(LOG_ERR, "socket: %m");
                exit(1);
        }
        if (dup2(fd, 0) < 0) {
                syslog(LOG_ERR, "dup2: %m");
                exit(1);
        }
        (void) close(fd);
        if (listen(0, 5) < 0) {
                syslog(LOG_ERR, "listen: %m");
                exit(1);
        }
        salen = sizeof sa;
        if (getsockname(0, (struct sockaddr *) &sa, &salen) < 0) {
                syslog(LOG_ERR, "getsockname: %m");
                exit(1);
        }
        fprintf(stderr, "Data manager waiting on port %u\n",
                ntohs(sa.sin_port));
}

/*
 * check_socket:
 *      Make sure the socket we are listening on is okay
 */
static
int
check_socket()
{
#ifdef reserved_port_authentication
        struct sockaddr_in      sa;
        int                     salen;
```

dm/dm.c

```
#ifndef ntohs
        unsigned short          ntohs();
#endif
#endif
        UIDTYPE                 getuid();

        if (listen(0, 5) < 0) {
                syslog(LOG_ERR, "listen: %m");
                exit(1);
        }
        if (getuid() != 0)
                return 0;
#ifdef reserved_port_authentication
        salen = sizeof sa;
        if (getsockname(0, (struct sockaddr *) &sa, &salen) < 0) {
                syslog(LOG_ERR, "getsockname: %m");
                exit(1);
        }
        if (ntohs(sa.sin_port) >= IPPORT_RESERVED)
                return -1;
#endif
        return 0;
}
```

dm/dm_server.h

```
/*
 * $Header: dm_server.h,v 1.1 88/07/25 13:15:25 conrad Exp $
 * $Log:           dm_server.h,v $
 * Revision 1.1  88/07/25  13:15:25  conrad
 * Initial revision
 *
 */

#ifndef   DM_INCLUDE

#define   DM_A_EXIST      0
#define   DM_A_FILLED     1
#define   DM_A_FULL       2

#define   DM_I_MAX        0
#define   DM_I_FILLED     1

#define   DM_READLOCK     0
#define   DM_WRITELOCK    1

#define   DM_S_INT1       0
#define   DM_S_INT2       1
#define   DM_S_INT4       2
#define   DM_S_INT8       3
#define   DM_S_FLOAT4     4
#define   DM_S_FLOAT8     5
#define   DM_S_FLOAT16    6
#define   DM_S_CHAR       7
#define   DM_S_STRING     8
#define   DM_S_BYTE       9
#define   DM_S_TABLE      10
#define   DM_S_NTYPE      11

/*
 * Some constants for simplifying implementation
 */
#define   DM_TABLE_NAMELEN      32
#define   DM_ATTR_NAMELEN       32

#define   DM_INCLUDE
#endif
```

dm/error.c

```
/*
 * $Header: error.c,v 1.1 88/07/25 13:15:25 conrad Exp $
 * $Log:		error.c,v $
 * Revision 1.1  88/07/25  13:15:25  conrad
 * Initial revision
 *
 */

#include "error.h"
#include "reply.h"

/*
 * error_internal:
 *	Internal server error.  Clear out all clients.
 */
void
error_internal(msg)
char	*msg;
{
	static int		die = FALSE;

	if (die) {
		/* Re-entered from reply().  All bets are off now. */
		abort();
	}
	die = TRUE;
	while (clist != NULL) {
		reply(CL_FILENO(clist), RS_FAILED, RT_OTHER, 3, "SERVER E
		reply(CL_FILENO(clist), RS_FAILED, RT_OTHER, 3, msg);
		reply(CL_FILENO(clist), RS_FAILED, RT_OTHER, 0, "Bye");
		reply_end(CL_FILENO(clist));
		(void) cl_free(clist);
	}
	abort();
	/* NOTREACHED */
}

/*
 * error_handshake:
 *	Protocol handshaking screwed up.  Terminate single client.
 */
void
error_handshake(clp)
CLIENT	*clp;
{
	reply(CL_FILENO(clp), RS_INFO, RT_OTHER, 4, "Handshake error");
	reply(CL_FILENO(clp), RS_FAILED, RT_OTHER, 0, "Bye");
		reply_end(CL_FILENO(clp));
		(void) cl_free(clp);
}
```

dm/error.h

```
/*
 * $Header: error.h,v 1.1 88/07/25 13:15:26 pett Locked $
 * $Log:            error.h,v $
 * Revision 1.1  88/07/25  13:15:26  conrad
 * Initial revision
 *
 */

#ifndef   ERROR_INCLUDE

#include "config.h"
#include "client.h"

/*
 * List of access functions
 */
void      error_internal();                /* Internal server error */
void      error_handshake();               /* Protocol handshake error */

#define   ERROR_INCLUDE
#endif
```

dm/help.c

```
/*
 * $Header: help.c,v 1.3 88/09/22 15:49:26 conrad Exp $
 * $Log:          help.c,v $
 * Revision 1.3  88/09/22  15:49:26  conrad
 * Command line is already tokenized, just use results
 *
 * Revision 1.2  88/08/08  09:46:33  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.1  88/07/25  13:15:26  conrad
 * Initial revision
 *
 */

#include "client.h"
#include "reply.h"
#include <ctype.h>

static char        *help[] = {
        "List of commands:",
        "        VERS    HELO    USER    MACHINE",
        "        OPEN    CLOSE   CREATE  DELETE",
        "        READ    WRITE   EMPTY   LOCKMODE",
        "        LOCK    UNLOCK  COUNT   LIST",
        "        QUIT    DESCRIBE",
        "For more info, type \"HELP <command>\"",
        NULL,
};

void    h_vers(), h_helo(), h_user(), h_quit(), h_machine();
void    h_open(), h_create(), h_delete(), h_close();
void    h_read(), h_write(), h_lock(), h_unlock(), h_lockmode();
void    h_count(), h_list(), h_empty(), h_describe();

typedef struct help_list {
        char    *name;
        void    (*func)();
}       HELP;

static HELP        help_list[] = {
        "VERS",         h_vers,
        "HELO",         h_helo,
        "USER",         h_user,
        "QUIT",         h_quit,
        "MACHINE",      h_machine,
        "OPEN",         h_open,
        "CREATE",       h_create,
        "DELETE",       h_delete,
        "CLOSE",        h_close,
        "READ",         h_read,
        "WRITE",        h_write,
        "LOCK",         h_lock,
        "UNLOCK",       h_unlock,
        "LOCKMODE",     h_lockmode,
        "COUNT",        h_count,
        "LIST",         h_list,
        "EMPTY",        h_empty,
        "DESCRIBE",     h_describe,
        NULL,           NULL,
};

/*
 * cl_help:
 *      Print the help msg to a client
 */
void
cl_help(clp)
CLIENT  *clp;
{
        register char   **cp;
        register HELP   *hp;
        register char   *ap;
        char            buf[BUFLEN];

        if (clp->nargs == 2) {
                /* See if there is a known command specified */
                for (ap = clp->args[1]; *ap != '\0'; ap++)
                        if (islower(*ap))
                                *ap = toupper(*ap);
                for (hp = help_list; hp->name != NULL; hp++)
                        if (strcmp(hp->name, clp->args[1]) == 0)
                                break;
                if (hp->name != NULL) {
                        (*hp->func)(clp);
                        reply_end(clp->fd);
                        return;
                }
                (void) sprintf(buf, "Unknown command \"%s\"", clp->args[1]);
                reply(clp->fd, RS_INFO, RT_OTHER, 0, buf);
        }
        /* Just list commands */
        for (cp = help; *cp != NULL; cp++)
                reply(clp->fd, RS_INFO, RT_OTHER, 0, *cp);
```

```
	reply_end(clp->fd);
}

/*
 * h_vers:
 *	Print help message for VERS command
 */
static
void
h_vers(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "VERS min max");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tClient speaks protocol versions between <min> and <max>");
}

/*
 * h_helo:
 *	Print help message for HELO command
 */
static
void
h_helo(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "HELO cid");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tClient conversation identifier is <cid>");
}

/*
 * h_user:
 *	Print help message for USER command
 */
static
void
h_user(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "USER local remote");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tClient invoked by <remote> to use account <local>");
}

/*
 * h_quit:
 *	Print help message for QUIT command
 */
static
void
h_quit(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "QUIT");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "\tClient wants to quit");
}

/*
 * h_machine:
 *	Print help message for MACHINE command
 */
static
void
h_machine(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "MACHINE <type>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "\tClient's machine is a <type>");
}

/*
 * h_open:
 *	Print help message for OPEN command
 */
static
void
h_open(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "OPEN DATASET <name>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "\tOpen dataset with <name>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "OPEN TABLE <name>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "\tOpen table with <name>");
}

/*
 * h_create:
 *	Print help message for CREATE command
 */
static
void
h_create(clp)
CLIENT	*clp;
```

```
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "CREATE DATASET <name>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "\tCreate dataset with <name>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "CREATE TABLE <name> <class>
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tCreate table with <name> and <class>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"CREATE ATTRIBUTE <table> <name> <dtype> <dlength>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tCreate attribute in <table> with <name> and data");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\ttype <dtype> and <dlength>>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"CREATE INSTANCE <table>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tCreate a new instance in <table>");
}

/*
 * h_delete:
 *	Print help message for DELETE command
 */
static
void
h_delete(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "DELETE DATASET <name>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "\tDelete dataset with <name>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "DELETE TABLE <name>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "\tDelete table with <name>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"DELETE ATTRIBUTE <table> <attribute>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "\tDelete <attribute> in <table>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "DELETE INSTANCE <table> <n>"
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tDelete instance <n> in <table>");
}

/*
 * h_close:
 *	Print help message for CLOSE command
 */
static
void
h_close(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "CLOSE TABLE <name>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "\tClose table with <name>");
}

/*
 * h_read:
 *	Print help message for READ command
 */
static
void
h_read(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"READ <table> <attribute> <from> <to>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tSend rows <from> thru <to> of attribute of table to client");
}

/*
 * h_write:
 *	Print help message for WRITE command
 */
static
void
h_write(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"WRITE <table> <attribute> <from> <to>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tFill rows <from> thru <to> of attribute of table with");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tforthcoming data");
}

/*
 * h_lock:
 *	Print help message for LOCK command
 */
static
void
h_lock(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
```

```
		"LOCK TABLE <table> <mode> <key> <wait>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"LOCK ATTRIBUTE <table> <attribute> <mode> <key> <wait>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tLock the specified item");
}

/*
 * h_lockmode:
 *	Print help message for LOCKMODE command
 */
static
void
h_lockmode(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"LOCKMODE TABLE <table> <mode> <key> <wait>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"LOCKMODE ATTRIBUTE <table> <attribute> <mode> <key> <w
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tChange the mode of the lock that client already holds");
}

/*
 * h_unlock:
 *	Print help message for UNLOCK command
 */
static
void
h_unlock(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"UNLOCK TABLE <table> <key>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"UNLOCK ATTRIBUTE <table> <attribute> <key>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tRelease the lock on the specified item");
}

/*
 * h_count:
 *	Print help message for COUNT command
 */
static
void
h_count(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "COUNT TABLE");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "COUNT TABLE CLASS <class>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tCount total number of tables or tables of given <class>");
}

/*
 * h_list:
 *	Print help message for LIST command
 */
static
void
h_list(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "LIST TABLE");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "LIST TABLE CLASS <class>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tCount total number of tables or tables of given <class>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0, "LIST ATTRIBUTE <table> <filled>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tList attributes of <table> of type <filled>");
}

/*
 * h_empty:
 *	Print help message for EMPTY command
 */
static
void
h_empty(clp)
CLIENT	*clp;
{
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"EMPTY <table> <attribute> <from> <to>");
	reply(clp->fd, RS_INFO, RT_OTHER, 0,
		"\tMark instances between <from> and <to> as empty");
}

/*
 * h_describe:
 *	Print help message for DESCRIBE command
 */
static
```

dm/help.c

```
void
h_describe(clp)
CLIENT  *clp;
{
        reply(clp->fd, RS_INFO, RT_OTHER, 0, "DESCRIBE TABLE <table>");
        reply(clp->fd, RS_INFO, RT_OTHER, 0,
                "DESCRIBE ATTRIBUTE <table> <attribute>");
        reply(clp->fd, RS_INFO, RT_OTHER, 0,
                "\tSend information about table or attribute");
}
```

dm/lock.c

```
/*
 * $Header: lock.c,v 1.5 89/03/02 20:59:18 pett Exp $
 * $Log:		lock.c,v $
 * Revision 1.5  89/03/02  20:59:18  pett
 * declared emalloc to return void *
 *
 * Revision 1.4  88/09/14  14:27:54  conrad
 * Change variable "lock" to "lock_list" so dbx doesn't choke
 *
 * Revision 1.3  88/08/08  09:46:35  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.2  88/07/28  09:39:28  conrad
 * Add new function for checking whether a lock is held by anyone
 * other than specified client
 *
 * Revision 1.1  88/07/25  13:15:27  conrad
 * Initial revision
 *
 */

#include "lock.h"
#include "error.h"

LOCK		*lock_list = NULL;
static long	lock_id = 1;

/*
 * lock_find:
 *	Find a lock (create it if necessary)
 */
LOCK *
lock_find(type, item)
int	type;
void	*item;
{
	register LOCK	*lp;
	extern void	*emalloc();

	for (lp = lock_list; lp != NULL; lp = lp->next)
		if (lp->type == type && lp->item == item)
			break;
	if (lp == NULL) {
		lp = (LOCK *) emalloc(sizeof (LOCK));
		lp->next = lock_list;
		lock_list = lp;
		lp->type = type;
		lp->item = item;
		lp->waiter = NULL;
		lp->holder = NULL;
		lp->key = lock_id++;
		lp->read_cnt = lp->write_cnt = 0;
	}

	return lp;
}

/*
 * lock_free:
 *	Destroy lock
 */
int
lock_free(target)
LOCK	*target;
{
	register LOCK	*lp, *prev;
	char		buf[BUFLEN], lbuf[BUFLEN];

	/*
	 * See if the lock is on our list
	 */
	prev = NULL;
	for (lp = lock_list; lp != NULL; lp = lp->next) {
		if (lp == target)
			break;
		prev = lp;
	}
	if (lp != target)
		return -1;

	/*
	 * Remove the lock from our list
	 */
	if (prev == NULL)
		lock_list = lp->next;
	else
		prev->next = lp->next;

	/*
	 * Release lock itself
	 */
	if (lp->waiter != NULL) {
		lock_name(lp, lbuf);
```

```
			(void) sprintf(buf, "Freeing busy lock %s", lbuf);
			error_internal(buf);
			/* NOTREACHED */
		}
		(void) free((char *) lp);
		return 0;
}

/*
 * lock_get:
 *		Try to get a lock
 */
lock_get(lp, mode, key, clp)
LOCK	*lp;
int	mode;
long	key;
CLIENT	*clp;
{
		register LHOLD	*lhp;
		register int	grant;
		extern void	*emalloc();

		if (lp->key == key)
			grant = TRUE;
		else if (key != 0)
			grant = FALSE;
		else if (lp->write_cnt > 0 || (lp->read_cnt > 0 && mode == LOCK_WRITE))
			grant = FALSE;
		else
			grant = TRUE;
		if (!grant)
			return -1;
		if (mode == LOCK_READ)
			lp->read_cnt++;
		else
			lp->write_cnt++;
		lhp = (LHOLD *) emalloc(sizeof (LHOLD));
		lhp->next = lp->holder;
		lp->holder = lhp;
		lhp->hold = clp;
		lhp->mode = mode;
		return lp->key;
}

/*
 * lock_release:
 *		Release a lock
 */
int
lock_release(lp, key, clp)
LOCK	*lp;
long	key;
CLIENT	*clp;
{
		register LHOLD	*lhp, *prev;
		register LWAIT	*lwp;

		/*
		 * Make sure that we have permission to release the lock
		 */
		if (key != 0 && lp->key != key)
			return -1;

		/*
		 * Remove the holder from our holder list
		 */
		prev = NULL;
		for (lhp = lp->holder; lhp != NULL; lhp = lhp->next) {
			if (lhp->hold == clp)
				break;
			prev = lhp;
		}
		if (lhp == NULL)
			return -1;
		if (prev == NULL)
			lp->holder = lhp->next;
		else
			prev->next = lhp->next;

		/*
		 * Update lock status then check the first waiter on the queue
		 * to see if it can acquire the lock.
		 */
		if (lhp->mode == LOCK_READ)
			lp->read_cnt--;
		else
			lp->write_cnt--;
		(void) free((char *) lhp);
		while (lp->waiter != NULL) {
			lwp = lp->waiter;
			lp->waiter = lwp->next;
			if (lp->write_cnt != 0)
				break;
			if (lwp->mode == LOCK_READ || lp->read_cnt == 0)
```

```
				(*lwp->f)(lp, lwp->mode, lwp->wait);
			else if (lwp->mode == LOCK_UPGRADE && lp->read_cnt == 1)
				(*lwp->f)(lp, lwp->mode, lwp->wait);
			(void) free((char *) lwp);
		}
		return 0;
}

/*
 * lock_wait:
 *	Put client on the wait queue for a particular lock
 */
void
lock_wait(lp, mode, clp, func)
LOCK	*lp;
int	mode;
CLIENT	*clp;
int	(*func)();
{
	register LWAIT	*lwp, *prev;
	extern void	*emalloc();

	lwp = (LWAIT *) emalloc(sizeof (LWAIT));
	lwp->f = func;
	lwp->wait = clp;
	lwp->mode = mode;
	lwp->next = NULL;
	if (lp->waiter == NULL)
		lp->waiter = lwp;
	else {
		for (prev = lp->waiter; prev->next != NULL; prev = prev->next)
			continue;
		prev->next = lwp;
	}
}

/*
 * lock_mode:
 *	Return the lock mode of the client
 */
int
lock_mode(lp, clp)
LOCK	*lp;
CLIENT	*clp;
{
	register LHOLD	*lhp;

	for (lhp = lp->holder; lhp != NULL; lhp = lhp->next)
		if (lhp->hold == clp)
			return lhp->mode;
	return LOCK_NONE;
}

/*
 * lock_by_other:
 *	Return whether this attribute is locked for reading or
 *	writing by anyone other than the client
 */
int
lock_by_other(lp, clp)
LOCK	*lp;
CLIENT	*clp;
{
	register LHOLD	*lhp;

	for (lhp = lp->holder; lhp != NULL; lhp = lhp->next)
		if (lhp->hold != clp || lhp->mode == LOCK_READ)
			return -1;
	return 0;
}

/*
 * lock_chmod:
 *	Change the mode of a lock (if permissible)
 */
int
lock_chmod(lp, mode, key, clp)
LOCK	*lp;
int	mode;
long	*key;
CLIENT	*clp;
{
	register LHOLD	*lhp;

	if (*key != 0 && lp->key != *key)
		return -1;
	for (lhp = lp->holder; lhp != NULL; lhp = lhp->next)
		if (lhp->hold == clp)
			break;
	if (lhp == NULL)
		return -1;
	if (lhp->mode == mode) {
		*key = lp->key;
		return 0;
```

```
	}
	if (lhp->mode == LOCK_WRITE) {
		/* Going from write-lock to read-lock is always safe */
		lhp->mode = LOCK_READ;
		lp->read_cnt++;
		lp->write_cnt--;
		*key = lp->key;
		return 0;
	}
	else {
		/* Going from read-lock to write-lock is not */
		if (lp->read_cnt == 1 && lp->write_cnt == 0) {
			lhp->mode = LOCK_WRITE;
			lp->read_cnt--;
			lp->write_cnt++;
			*key = lp->key;
			return 0;
		}
		else
			return 1;
	}
}

/*
 * lock_remove_client:
 *	If client holds any locks, release them.
 *	If client is waiting for a lock, dequeue him.
 */
void
lock_remove_client(clp)
CLIENT	*clp;
{
	register LOCK	*lp;
	register LWAIT	*lwp, *prev;

	for (lp = lock_list; lp != NULL; lp = lp->next) {
		while (lock_release(lp, lp->key, clp) == 0)
			continue;
		prev = NULL;
		for (lwp = lp->waiter; lwp != NULL; lwp = lwp->next) {
			if (lwp->wait == clp)
				break;
			prev = lwp;
		}
		if (lwp->wait == clp) {
			if (prev == NULL)
				lp->waiter = lwp->next;
			else
				prev->next = lwp->next;
		}
	}
}

/*
 * lock_name:
 *	Construct a name for a lock
 */
void
lock_name(lp, buf)
LOCK	*lp;
char	*buf;
{
	(void) sprintf(buf, "lock[%ld,%d]", lp->key, lp->type);
}
```

dm/lock.h

```
/*
 * $Header: lock.h,v 1.3 88/08/08 09:46:36 conrad Exp $
 * $Log:        lock.h,v $
 * Revision 1.3  88/08/08  09:46:36  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.2  88/07/28  09:39:03  conrad
 * Add new function for checking whether a lock is held by anyone
 * other than specified client
 *
 * Revision 1.1  88/07/25  13:15:28  conrad
 * Initial revision
 *
 */

#ifndef LOCK_INCLUDE

#include "config.h"
#include "client.h"

/*
 * How locks may be locked
 */
#define LOCK_NONE      -1
#define LOCK_READE     0
#define LOCK_WRITE     1
#define LOCK_UPGRADE   2

/*
 * Data structure
 */
typedef struct lwait          {
        struct lwait          *next;
        int                   (*f)();
        CLIENT                *wait;
        int                   mode;
}       LWAIT;

typedef struct lhold          {
        struct lhold          *next;
        CLIENT                *hold;
        int                   mode;
}       LHOLD;

typedef struct lock           {
        struct lock           *next;
        long                  key;
        int                   read_cnt;
        int                   write_cnt;
        int                   type;
        void                  *item;
        LWAIT                 *waiter;
        LHOLD                 *holder;
}       LOCK;

/*
 * Package variables
 */
extern LOCK     *lock;

/*
 * List of access functions
 */
LOCK    *lock_find();             /* Find a lock (create if necessary) */
int     lock_free();              /* Release old lock */
int     lock_get();               /* Get a lock */
int     lock_release();           /* Release a lock */
void    lock_wait();              /* Wait for a lock */
void    lock_remove_client();     /* Cleanup locks held by client */
int     lock_mode();              /* Return mode of lock */
int     lock_by_other();          /* Check if anyone else holds lock */
void    lock_name();              /* Construct name of lock */

#define LOCK_INCLUDE
#endif
```

dm/misc.c

```
/*
 * $Header: misc.c,v 1.3 89/03/02 22:01:02 pett Exp $
 * $Log:          misc.c,v $
 * Revision 1.3  89/03/02  22:01:02  pett
 * declared emalloc/erealloc to return void *
 *
 * Revision 1.2  88/08/04  16:26:31  conrad
 * Support data type STRING and TABLE
 *
 * Revision 1.1  88/07/25  13:15:28  conrad
 * Initial revision
 *
 */

#include "error.h"

/*
 * copy_string:
 *	Make a copy of the given string
 */
char *
copy_string(s)
char	*s;
{
	register char	*cp;
	extern void	*emalloc();
	extern char	*strcpy();

	cp = (char *) emalloc(strlen(s) + 1);
	(void) strcpy(cp, s);
	return cp;
}

/*
 * emalloc:
 *	Exit-on-error malloc
 */
void *
emalloc(n)
int	n;
{
	register void		*cp;
	extern MALLOCTYPE	malloc();

	if ((cp = (void *) malloc((unsigned) n)) == NULL)
		error_internal("emalloc");
	return cp;
}

/*
 * erealloc:
 *	Exit-on-error realloc
 */
void *
erealloc(ptr, n)
void	*ptr;
int	n;
{
	register void		*cp;
	extern void		*emalloc();
	extern MALLOCTYPE	realloc();

	if (ptr == NULL)
		return emalloc(n);
	if ((cp = (void *) realloc((MALLOCTYPE) ptr, (unsigned) n)) == NULL)
		error_internal("erealloc");
	return cp;
}
```

dm/parse.c

```c
/*
 * $Header: parse.c,v 1.13 89/02/23 17:57:30 pett Exp $
 * $Log:		parse.c,v $
 * Revision 1.13  89/02/23  17:57:30  pett
 * changed some declarations to take advantage of machdep.h
 *
 * Revision 1.12  88/10/06  12:45:48  conrad
 * Add statistics gathering
 *
 * Revision 1.11  88/10/04  15:04:31  conrad
 * Handle inputs as true streams with no record boundaries (newlines are
 * no longer expected as the last character of an input buffer)
 *
 * Revision 1.10  88/09/29  14:06:14  conrad
 * Handle input properly (i.e. multiple command in the same buffer now works)
 * EOF by clients waiting for a lock also handled properly now
 *
 * Revision 1.9  88/09/28  10:43:23  conrad
 * Implement multiple groups if NGROUPS is defined
 *
 * Revision 1.8  88/09/19  17:13:06  conrad
 * Use tokenize() for all commands including initial handshake
 *
 * Revision 1.7  88/09/14  14:29:38  conrad
 * Fix attribute unlocking
 *
 * Revision 1.6  88/09/07  14:53:29  conrad
 * Treat EOF from client as "normal" rather than error
 *
 * Revision 1.5  88/08/08  09:46:37  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.4  88/07/29  15:01:21  conrad
 * Fix protocol botch
 *
 * Revision 1.3  88/07/27  20:24:31  conrad
 * Use reply parameter instead of hardwired number
 *
 * Revision 1.2  88/07/27  15:24:22  conrad
 * Don't bother sending out "Please wait" messages
 *
 * Revision 1.1  88/07/25  13:15:29  conrad
 * Initial revision
 *
 */

#include "client.h"
#include "reply.h"
#include "error.h"
#include "dataset.h"
#include "table.h"
#include <string.h>

#define	HANDSHAKE_MIN 1
#define	HANDSHAKE_MAX1
#define	MIN_VERSION	1
#define	MAX_VERSION	1

/*
 * cl_process:
 *		This client has input.  Consume it.
 *		Return non-zero if we are at the end of input, zero otherwise.
 */
cl_process(clp, woke)
CLIENT	*clp;
int	woke;
{
	register char		*endp;

	/*
	 * If we are waiting for data, we just let the lower level
	 * routines handle input data
	 */
	if (clp->state == CS_DATAWAIT) {
		if (cell_data(clp) == 0) {
			(void) free((char *) clp->iov);
			clp->state = CS_DMCMD;
			reply_end(clp->fd);
		}
		return (clp->indx == 0);
	}

	/*
	 * If we just woke up from a LOCKWAIT, then the command
	 * is already tokenized and all, so we just skip all the
	 * checks and tokenizing that we normally need
	 */
	if (!woke) {
		if ((endp = strchr(clp->ibuf, '\n')) == NULL) {
			/* Incomplete command */
			return 1;
		}
		*endp++ = '\0';
```

```
		clp->combuf = clp->ibuf;
		clp->nargs = tokenize(clp->combuf, clp->args, COM_MAX_TOKE
	}

	/*
	 * If the command is obviously malformed, we don't even bother
	 * trying to parse it
	 */
	if (clp->nargs == 0 || clp->nargs == -2 || clp->nargs == -3) {
		reply(clp->fd, RS_REJECT, RT_OTHER, 0, "Malformed command
		reply_end(clp->fd);
	}
	else
		(void) cl_parse(clp);

	/*
	 * If we went into a CS_LOCKWAIT state, then we claim we are at
	 * end of input even if we aren't.  We also do not consume the
	 * input since it still needs to be re-parsed.  Otherwise, we
	 * consume the command and check whether input remains.
	 */
	if (clp->state != CS_LOCKWAIT) {
		cl_consume(clp, endp - clp->ibuf);
		return (clp->indx == 0);
	}
	else
		return 1;
}

/*
 * cl_consume:
 *	Consume n bytes from input buffer
 */
void
cl_consume(clp, nbytes)
CLIENT	*clp;
int	nbytes;
{
	register char	*lastp;
	register char	*endp, *cp;

	if (nbytes <= 0)
		return;
	lastp = &clp->ibuf[clp->indx];
	endp = &clp->ibuf[nbytes];
	clp->indx = lastp - endp;
	if (clp->indx == 0)
		return;
#ifdef DEBUG
	if (clp->indx < 0)
		error_internal("negative buffer index");
#endif
	cp = clp->ibuf;
	while (endp < lastp)
		*cp++ = *endp++;
	*cp = '\0';
}

/*
 * cl_parse:
 *	Get client to parse command
 */
cl_parse(clp)
CLIENT	*clp;
{
	int	vers(), helo(), user(), cl_dmcmd();
	int	tokenize();

	if (strcmp(clp->args[0], "HELP") == 0) {
		cl_help(clp);
		return 0;
	}
	switch (clp->state) {
	 case CS_VERS:
		if (vers(clp) < 0)
			goto error;
		break;
	 case CS_HELO:
		if (helo(clp) < 0)
			goto error;
		break;
	 case CS_USER:
		if (user(clp) < 0)
			goto error;
		break;
	 case CS_MACHINE:
		if (machine(clp) < 0)
			goto error;
		break;
	 case CS_DMCMD:
		if (cl_dmcmd(clp) < 0)		/* Error handling done already */
			return -1;
		break;
	}
```

```
	return 0;

error:
	error_handshake(clp);
	return -1;
}

/*
 * vers:
 *	Version number exchange
 */
static
vers(clp)
CLIENT	*clp;
{
	int	hmin, hmax;
	int	min, max;
	int	hversion;
	int	version;
	char	msg_buf[BUFLEN];

	if (clp->nargs != 5) {
		reply(clp->fd, RS_REJECT, RT_OTHER, 1, "Command syntax err
		return -1;
	}
	if (strcmp(clp->args[0], "VERS") != 0) {
		reply(clp->fd, RS_REJECT, RT_OTHER, 0, "VERS command mis
		return -1;
	}
	hmin = atoi(clp->args[1]);
	hmax = atoi(clp->args[2]);
	min = atoi(clp->args[3]);
	max = atoi(clp->args[4]);
	if (HANDSHAKE_MIN > hmax || HANDSHAKE_MAX < hmin) {
		reply(clp->fd, RS_FAILED, RT_OTHER, 1,
			"Protocol version mismatch");
		return -1;
	}
	if (MIN_VERSION > max || MAX_VERSION < min) {
		reply(clp->fd, RS_FAILED, RT_OTHER, 1,
			"Protocol version mismatch");
		return -1;
	}
	hversion = (HANDSHAKE_MAX > hmax) ? hmax : HANDSHAKE_MAX;
	version = (MAX_VERSION > max) ? max : MAX_VERSION;
	reply(clp->fd, RS_ACCEPT, RT_OTHER, 0, "Okay");
	reply_begin_ASCII(clp->fd);
	(void) sprintf(msg_buf, "%d %d", hversion, version);
	reply_ASCII(clp->fd, msg_buf);
	reply_end_ASCII(clp->fd);
	reply_end(clp->fd);
	clp->state = CS_HELO;
	return 0;
}

/*
 * helo:
 *	Conversation ID exchange
 */
static
helo(clp)
CLIENT	*clp;
{
	extern int		cid_verify();

	if (clp->nargs != 2) {
		reply(clp->fd, RS_REJECT, RT_OTHER, 1, "Command syntax error");
		return -1;
	}
	if (strcmp(clp->args[0], "HELO") != 0) {
		reply(clp->fd, RS_REJECT, RT_OTHER, 0, "HELO command missing");
		return -1;
	}
	if (cid_verify(clp->args[1]) < 0) {
		reply_noperm(clp->fd);
		return -1;
	}
	reply(clp->fd, RS_ACCEPT, RT_OTHER, 0, "Command accepted");
	reply_end(clp->fd);
	clp->state = CS_USER;
	return 0;
}

/*
 * user:
 *	User verification
 */
static
user(clp)
CLIENT	*clp;
{
	int	verify_user();

	if (clp->nargs != 3) {
```

```
		reply(clp->fd, RS_REJECT, RT_OTHER, 1, "Command syntax err
		return -1;
	}
	if (strcmp(clp->args[0], "USER") != 0) {
		reply(clp->fd, RS_REJECT, RT_OTHER, 0, "USER command mis
		return -1;
	}
	if (verify_user(clp, clp->args[1], clp->args[2]) < 0) {
		reply_noperm(clp->fd);
		return -1;
	}
	reply(clp->fd, RS_ACCEPT, RT_OTHER, 0, "Command accepted");
	reply_end(clp->fd);
	clp->state = CS_MACHINE;
	return 0;
}

/*
 * machine:
 *	Machine type exchange
 */
static
machine(clp)
CLIENT	*clp;
{
	if (clp->nargs != 2) {
		reply(clp->fd, RS_REJECT, RT_OTHER, 1, "Command syntax err
		return -1;
	}
	if (strcmp(clp->args[0], "MACHINE") != 0) {
		reply(clp->fd, RS_REJECT, RT_OTHER, 0,
			"MACHINE command missing");
		return -1;
	}
	if (strcmp(clp->args[1], MACHINE_TYPE) != 0) {
		reply(clp->fd, RS_ACCEPT, RT_DM, RG_ASCII, "Use ASCII data
		clp->data_mode = REPLY_IN_ASCII;
	}
	else {
		reply(clp->fd, RS_ACCEPT, RT_DM, RG_BINARY, "Use binary d
		clp->data_mode = REPLY_IN_BINARY;
	}
	reply_end(clp->fd);
	clp->state = CS_DMCMD;
	return 0;
}
```

```
#include <netdb.h>
#include <pwd.h>
#include <grp.h>

/*
 * verify_user:
 *	Make sure that the remote account has access to the local account
 */
static
int
verify_user(clp, local, remote)
CLIENT	*clp;
char	*local, *remote;
{
	register char	*host;
	register int	status;
	struct hostent	*hp;
	struct passwd	*pp;
	void		group_init();

	host = strchr(remote, '@');
	if (host == NULL)
		return -1;
	pp = getpwnam(local);
	if (pp == NULL)
		return -1;
	if (chdir(pp->pw_dir) < 0)
		return -1;
	*host++ = '\0';
	hp = gethostbyname(host);
	if (hp == NULL)
		return -1;
	status = ruserok(hp->h_name, strcmp(local, "root") == 0, remote, local);
	if (ds_visit() < 0) {
		error_internal("Can't get back to dataset directory");
		/* NOTREACHED */
	}
	clp->uid = pp->pw_uid;
#ifdef NGROUPS
	group_init(clp, local, pp->pw_gid);
#else
	clp->gid = pp->pw_gid;
#endif
	return status;
}

#ifdef NGROUPS
```

dm/parse.c

```c
/*
 * group_init:
 *	Initialize list of groups
 */
static
void
group_init(clp, acct, base_group)
CLIENT	*clp;
char	*acct;
int	base_group;
{
	register struct group		*grp;
	register int			i;

	clp->ngroups = 0;
	if (base_group >= 0)
		clp->groups[clp->ngroups++] = base_group;
	(void) setgrent();
	while ((grp = getgrent()) != NULL) {
		if (grp->gr_gid == base_group)
			continue;
		for (i = 0; grp->gr_mem[i]; i++) {
			if (strcmp(grp->gr_mem[i], acct) != 0)
				continue;
			clp->groups[clp->ngroups++] = grp->gr_gid;
			if (clp->ngroups >= NGROUPS) {
				endgrent();
				return;
			}
		}
	}
	endgrent();
}
#endif

typedef struct com_list {
	char	*command;
	int	(*func)();
}	COM_LIST;

int	c_quit();
int	c_open(), c_create(), c_delete(), c_close();
int	c_read(), c_write(), c_lock(), c_unlock(), c_lockmode();
int	c_count(), c_list(), c_empty(), c_describe();

static COM_LIST	com_list[] = {
	"QUIT",		c_quit,
	"OPEN",		c_open,
	"CREATE",	c_create,
	"DELETE",	c_delete,
	"CLOSE",	c_close,
	"READ",		c_read,
	"WRITE",	c_write,
	"LOCK",		c_lock,
	"LOCKMODE",	c_lockmode,
	"UNLOCK",	c_unlock,
	"COUNT",	c_count,
	"LIST",		c_list,
	"EMPTY",	c_empty,
	"DESCRIBE",	c_describe,
	NULL,		NULL,
};

/*
 * cl_dmcmd:
 *	Parse a data manager command
 */
cl_dmcmd(clp)
CLIENT	*clp;
{
	register COM_LIST		*cp;
#ifdef STATS
	extern int		ntotal_com;

	ntotal_com++;
#endif
	for (cp = com_list; cp->command != NULL; cp++)
		if (strcmp(cp->command, clp->args[0]) == 0)
			break;
	if (cp->command == NULL) {
		reply(clp->fd, RS_REJECT, RT_OTHER, 0, "Unknown command");
		reply_end(clp->fd);
		return 0;
	}
	if (clp->nargs == -1) {
		reply(clp->fd, RS_REJECT, RT_OTHER, 1, "Too many arguments");
		reply_end(clp->fd);
		return 0;
	}
	return (*cp->func)(clp);
}

#if 0
/*
```

```
 * c_unimplemented:
 *         Inform user that this command is unimplemented
 */
static
int
c_unimplemented(clp)
CLIENT  *clp;
{
        reply(clp->fd, RS_FAILED, RT_DM, 9, "Command is unimplemented");
        reply_end(clp->fd);
        return 0;
}
#endif

/*
 * c_badcount:
 *         Inform user that he gave the wrong number of arguments
 */
static
void
c_badcount(clp)
CLIENT  *clp;
{
        reply(clp->fd, RS_REJECT, RT_OTHER, 1, "Wrong number of arguments");
        reply_end(clp->fd);
}

/*
 * c_syntax:
 *         Report a syntax error
 */
static
void
c_syntax(clp)
CLIENT  *clp;
{
        reply(clp->fd, RS_REJECT, RT_OTHER, 1, "Syntax error");
        reply_end(clp->fd);
}

/*
 * c_quit:
 *         Execute QUIT command
 */
static
int
c_quit(clp)
CLIENT  *clp;
{
        reply(clp->fd, RS_ACCEPT, RT_OTHER, 0, "Bye");
        reply_end(clp->fd);
        (void) cl_free(clp);
        return -1;
}

/*
 * c_open:
 *         Execute OPEN command
 */
static
int
c_open(clp)
CLIENT  *clp;
{
        char    name[DS_NAMESIZE];

        if (clp->nargs != 3)
                c_badcount(clp);
        else if (strcmp(clp->args[1], "DATASET") == 0) {
                /*
                 * Must use temporary buffer in case ds_open tries to
                 * tell us name of dataset that another data manager
                 * is using.
                 */
                (void) strcpy(name, clp->args[2]);
                switch (ds_open(name, clp)) {
                 case DS_READWRITE:
                        reply(clp->fd, RS_ACCEPT, RT_DM, RG_RW,
                                "Dataset opened for reading and writing");
                        break;
                 case DS_READONLY:
                        reply(clp->fd, RS_ACCEPT, RT_DM, RG_RO,
                                "Dataset opened for reading only");
                        break;
                 case DS_NOFILE:
                        reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
                                "No such dataset");
                        break;
                 case DS_NOPERM:
                        reply_noperm(clp->fd);
                        break;
                 case DS_UNAVAIL:
                        reply(clp->fd, RS_FAILED, RT_DM, RB_UNAVAIL,
                                "Data set already opened by another DM");
```

```c
				reply_begin_ASCII(clp->fd);
				reply_ASCII(clp->fd, name);
				reply_end_ASCII(clp->fd);
				break;
			 case DS_WRONGFILE:
			 default:
				error_internal("Open dataset failed");
				/* NOTREACHED */
			}
			reply_end(clp->fd);
		}
		else if (strcmp(clp->args[1], "TABLE") == 0) {
			switch (tbl_open(clp, clp->args[2])) {
			 case TBL_SUCCESS:
				reply(clp->fd, RS_ACCEPT, RT_DM, RG_OKAY,
					"Table opened");
				break;
			 case TBL_NOSUCH:
				reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
					"No such table");
				break;
			 default:
				error_internal("Open table failed");
				/* NOTREACHED */
			}
			reply_end(clp->fd);
		}
		else
			c_syntax(clp);
		return 0;
}

/*
 * c_close:
 *	Execute CLOSE command
 */
static
int
c_close(clp)
CLIENT	*clp;
{
		if (clp->nargs != 3)
			c_badcount(clp);
		else if (strcmp(clp->args[1], "TABLE") != 0)
			c_syntax(clp);
		else {
			switch (tbl_close(clp, clp->args[2])) {
			 case TBL_SUCCESS:
				reply(clp->fd, RS_ACCEPT, RT_DM, RG_OKAY,
					"Table closed");
				break;
			 case TBL_NOSUCH:
				reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
					"No such table");
				break;
			 case TBL_NOPERM:
				reply(clp->fd, RS_FAILED, RT_DM, RB_UNAVAIL,
					"Table was not opened");
				break;
			 default:
				error_internal("Close table failed");
				/* NOTREACHED */
			}
			reply_end(clp->fd);
		}
		return 0;
}

/*
 * c_create:
 *	Execute CREATE command
 */
static
int
c_create(clp)
CLIENT	*clp;
{
		char	*namep;
		int	class;
		int	dtype, dsize;
		int	inum;
		char	buf[BUFLEN];

		if (clp->nargs < 2)
			c_badcount(clp);
		else if (strcmp(clp->args[1], "DATASET") == 0) {
			if (clp->nargs != 3)
				c_badcount(clp);
			else {
				namep = clp->args[2];
				switch (ds_create(namep, clp)) {
				 case DS_EXISTS:
					reply(clp->fd, RS_FAILED, RT_DM, RB_EXISTS,
						"Dataset exists");
```

```
					break;
				case DS_NOPERM:
					reply_noperm(clp->fd);
					break;
				case DS_SUCCESS:
					reply(clp->fd, RS_ACCEPT, RT_DM, RG_OK
						"Dataset created");
					break;
				default:
					error_internal("Create dataset failed");
					/* NOTREACHED */
				}
				reply_end(clp->fd);
			}
		}
		else if (strcmp(clp->args[1], "TABLE") == 0) {
			if (clp->nargs != 4)
				c_badcount(clp);
			else {
				namep = clp->args[2];
				class = atoi(clp->args[3]);
				switch (tbl_new(clp, namep, class)) {
				case TBL_NOPERM:
					reply_noperm(clp->fd);
					break;
				case TBL_SUCCESS:
					reply(clp->fd, RS_ACCEPT, RT_DM, RG_OK
						"Table created");
					break;
				case TBL_EXISTS:
					reply(clp->fd, RS_FAILED, RT_DM, RB_EXIS
						"Table already exists");
					break;
				default:
					error_internal("Create table failed");
					/* NOTREACHED */
				}
				reply_end(clp->fd);
			}
		}
		else if (strcmp(clp->args[1], "ATTRIBUTE") == 0) {
			if (clp->nargs != 6)
				c_badcount(clp);
			else {
				dtype = atoi(clp->args[4]);
				dsize = atoi(clp->args[5]);
				switch (attr_new(clp, clp->args[2], clp->args[3],
				dtype, dsize)) {
				case ATTR_UNSUPPORTED:
					reply(clp->fd, RS_FAILED, RT_DM, RB_UNSUPPORTED,
						"Unsupported data type");
					break;
				case ATTR_SUCCESS:
					reply(clp->fd, RS_ACCEPT, RT_DM, RG_OKAY,
						"Attribute created");
					break;
				case TBL_NOSUCH:
					reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
						"No such table");
					break;
				case ATTR_EXISTS:
					reply(clp->fd, RS_FAILED, RT_DM, RB_EXISTS,
						"Attribute already exists");
					break;
				case TBL_NOPERM:
					reply_noperm(clp->fd);
					break;
				default:
					error_internal("Create attribute failed");
					/* NOTREACHED */
				}
				reply_end(clp->fd);
			}
		}
		else if (strcmp(clp->args[1], "INSTANCE") == 0) {
			if (clp->nargs != 3)
				c_badcount(clp);
			else {
				switch (inst_new(clp, clp->args[2], &inum)) {
				case TBL_NOSUCH:
					reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
						"No such table");
					break;
				case TBL_NOPERM:
				case ATTR_NOPERM:
					reply_noperm(clp->fd);
					break;
				case INST_SUCCESS:
					(void) sprintf(buf,
						"%d is new instance number", inum);
					reply(clp->fd, RS_ACCEPT, RT_DM, RG_OKAY, buf);
					break;
				default:
					error_internal("Create instance failed");
```

```
					/* NOTREACHED */
				}
				reply_end(clp->fd);
			}
		}
		else
			c_syntax(clp);
		return 0;
}

/*
 * c_delete:
 *	Execute DELETE command
 */
static
int
c_delete(clp)
CLIENT	*clp;
{
	char	*namep;
	int	inum;

	if (strcmp(clp->args[1], "DATASET") == 0) {
		if (clp->nargs != 3)
			c_badcount(clp);
		else {
			namep = clp->args[2];
			switch (ds_delete(namep, clp)) {
			 case DS_NOPERM:
				reply_noperm(clp->fd);
				break;
			 case DS_SUCCESS:
				reply(clp->fd, RS_ACCEPT, RT_DM, RG_OK
					"Dataset deleted");
				break;
			 case DS_WRONGFILE:
			 default:
				error_internal("Delete dataset failed");
				/* NOTREACHED */
			}
			reply_end(clp->fd);
		}
	}
	else if (strcmp(clp->args[1], "TABLE") == 0) {
		if (clp->nargs != 3)
			c_badcount(clp);
		else {
			switch (tbl_delete(clp, clp->args[2])) {
			 case TBL_NOPERM:
			 case ATTR_NOPERM:
				reply_noperm(clp->fd);
				break;
			 case TBL_NOSUCH:
				reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
					"No such table");
				break;
			 case TBL_SUCCESS:
				reply(clp->fd, RS_ACCEPT, RT_DM, RG_OKAY,
					"Table deleted");
				break;
			 default:
				error_internal("Delete table failed");
				/* NOTREACHED */
			}
			reply_end(clp->fd);
		}
	}
	else if (strcmp(clp->args[1], "ATTRIBUTE") == 0) {
		if (clp->nargs != 4)
			c_badcount(clp);
		else {
			switch (attr_delete(clp, clp->args[2], clp->args[3])) {
			 case TBL_NOPERM:
				reply_noperm(clp->fd);
				break;
			 case TBL_NOSUCH:
				reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
					"No such table");
				break;
			 case ATTR_NOSUCH:
				reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
					"No such attribute");
				break;
			 case ATTR_SUCCESS:
				reply(clp->fd, RS_ACCEPT, RT_DM, RG_OKAY,
					"Attribute deleted");
				break;
			 default:
				error_internal("Delete attribute failed");
				/* NOTREACHED */
			}
			reply_end(clp->fd);
		}
	}
```

```
	else if (strcmp(clp->args[1], "INSTANCE") == 0) {
		if (clp->nargs != 4)
			c_badcount(clp);
		else {
			inum = atoi(clp->args[3]);
			switch (inst_delete(clp, clp->args[2], inum)) {
			 case TBL_NOSUCH:
				reply(clp->fd, RS_FAILED, RT_DM, RB_NOS
					"No such table");
				break;
			 case TBL_NOPERM:
			 case ATTR_NOPERM:
				reply_noperm(clp->fd);
				break;
			 case INST_NOSUCH:
			 case INST_SUCCESS:
				reply(clp->fd, RS_ACCEPT, RT_DM, RG_OK
					"Instance deleted");
				break;
			 default:
				error_internal("Delete attribute failed");
				/* NOTREACHED */
			}
			reply_end(clp->fd);
		}
	}
	else
		c_syntax(clp);
	return 0;
}

/*
 * c_read:
 *	Execute READ command
 */
static
int
c_read(clp)
CLIENT	*clp;
{
	int	from, to;

	if (clp->nargs != 5) {
		c_badcount(clp);
		return 0;
	}
	from = atoi(clp->args[3]);
	to = atoi(clp->args[4]);
	switch (cell_read(clp, clp->args[1], clp->args[2], from, to)) {
	 case TBL_NOSUCH:
		reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH, "No such table");
		break;
	 case ATTR_NOSUCH:
		reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
			"No such attribute");
		break;
	 case ATTR_NOPERM:
		reply_noperm(clp->fd);
		break;
	 case CELL_SUCCESS:
		break;
	}
	reply_end(clp->fd);
	return 0;
}

/*
 * c_write:
 *	Execute WRITE command
 */
static
int
c_write(clp)
CLIENT	*clp;
{
	int	from, to;

	if (clp->nargs != 5) {
		c_badcount(clp);
		return 0;
	}
	from = atoi(clp->args[3]);
	to = atoi(clp->args[4]);
	switch (cell_write(clp, clp->args[1], clp->args[2], from, to)) {
	 case TBL_NOSUCH:
		reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH, "No such table");
		break;
	 case ATTR_NOSUCH:
		reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
			"No such attribute");
		break;
	 case ATTR_NOPERM:
		reply_noperm(clp->fd);
		break;
```

```
		case CELL_SUCCESS:
			reply(clp->fd, RS_MORE, RT_DM, RG_OKAY, "Send data");
			break;
		}
		reply_end(clp->fd);
		return 0;
}

/*
 * c_lock:
 *	Execute LOCK command
 */
static
int
c_lock(clp)
CLIENT	*clp;
{
	int	mode;
	long	key;
	int	wait;
	char	buf[BUFLEN];
#ifdef STATS
	extern int		nlock_com;

	nlock_com++;
#endif
	if (clp->nargs == 6) {
		if (strcmp(clp->args[1], "TABLE") == 0) {
			if (strcmp(clp->args[3], "read") == 0)
				mode = LOCK_READ;
			else
				mode = LOCK_WRITE;
			key = atoi(clp->args[4]);
			wait = atoi(clp->args[5]);
			switch (tbl_lock(clp, clp->args[2], mode, &key)) {
			  case TBL_SUCCESS:
				(void) sprintf(buf, "%ld is key", key);
				reply(clp->fd, RS_ACCEPT, RT_DM, RG_OK
				reply_end(clp->fd);
				break;
			  case TBL_LOCKWAIT:
				if (wait)
					tbl_sleep(clp, clp->args[2], mode);
				else {
					reply(clp->fd, RS_FAILED, RT_DM,
						RB_CANTLOCK,
						"Lock unavailable");
					reply_end(clp->fd);
				}
				break;
			  case TBL_NOSUCH:
				reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
					"No such table");
				reply_end(clp->fd);
				break;
			  case TBL_NOPERM:
				reply_noperm(clp->fd);
				reply_end(clp->fd);
				break;
			  default:
				error_internal("Lock table failed");
				/* NOTREACHED */
			}
		}
		else
			c_syntax(clp);
	}
	else if (clp->nargs == 7) {
		if (strcmp(clp->args[1], "ATTRIBUTE") == 0) {
			if (strcmp(clp->args[4], "read") == 0)
				mode = LOCK_READ;
			else
				mode = LOCK_WRITE;
			key = atoi(clp->args[5]);
			wait = atoi(clp->args[6]);
			switch (attr_lock(clp, clp->args[2],
			clp->args[3], mode, &key)) {
			  case ATTR_SUCCESS:
				(void) sprintf(buf, "%ld is key", key);
				reply(clp->fd, RS_ACCEPT, RT_DM, RG_OKAY, buf);
				reply_end(clp->fd);
				break;
			  case ATTR_LOCKWAIT:
				if (wait)
					attr_sleep(clp, clp->args[2],
						clp->args[3], mode);
				else {
					reply(clp->fd, RS_FAILED, RT_DM,
						RB_CANTLOCK,
						"Lock unavailable");
					reply_end(clp->fd);
				}
				break;
			  case TBL_NOSUCH:
```

```
                                        reply(clp->fd, RS_FAILED, RT_DM, RB_NOS
                                                "No such table");
                                        reply_end(clp->fd);
                                        break;
                                case ATTR_NOSUCH:
                                        reply(clp->fd, RS_FAILED, RT_DM, RB_NOS
                                                "No such attribute");
                                        reply_end(clp->fd);
                                        break;
                                case ATTR_NOPERM:
                                        reply_noperm(clp->fd);
                                        reply_end(clp->fd);
                                        break;
                                default:
                                        error_internal("Lock attribute failed");
                                        /* NOTREACHED */
                                }
                        }
                        else
                                c_syntax(clp);
                }
                else
                        c_badcount(clp);
                return 0;
}

/*
 * c_lockmode:
 *      Execute LOCKMODE command
 */
static
int
c_lockmode(clp)
CLIENT  *clp;
{
        int     mode;
        long    key;
        int     wait;
        char    buf[BUFLEN];
#ifdef STATS
        extern int      nlock_com;

        nlock_com++;
#endif
        if (clp->nargs == 6) {
                if (strcmp(clp->args[1], "TABLE") == 0) {
                        if (strcmp(clp->args[3], "read") == 0)
                                mode = LOCK_READ;
                        else
                                mode = LOCK_WRITE;
                        key = atoi(clp->args[4]);
                        wait = atoi(clp->args[5]);
                        switch (tbl_lockmode(clp, clp->args[2], mode, &key)) {
                         case TBL_SUCCESS:
                                (void) sprintf(buf, "%ld is key", key);
                                reply(clp->fd, RS_ACCEPT, RT_DM, RG_OKAY, buf);
                                reply_end(clp->fd);
                                break;
                         case TBL_LOCKWAIT:
                                if (wait)
                                        tbl_sleep(clp, clp->args[2],
                                                LOCK_UPGRADE);
                                else {
                                        reply(clp->fd, RS_FAILED, RT_DM,
                                                RB_CANTLOCK,
                                                "Lock unavailable");
                                        reply_end(clp->fd);
                                }
                                break;
                         case TBL_NOSUCH:
                                reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
                                        "No such table");
                                reply_end(clp->fd);
                                break;
                         case TBL_NOPERM:
                                reply(clp->fd, RS_FAILED, RT_DM, RB_BADKEY,
                                        "Key does not match lock");
                                reply_end(clp->fd);
                                break;
                         default:
                                error_internal("Lockmode table failed");
                                /* NOTREACHED */
                        }
                }
                else
                        c_syntax(clp);
        }
        else if (clp->nargs == 7) {
                if (strcmp(clp->args[1], "ATTRIBUTE") == 0) {
                        if (strcmp(clp->args[4], "read") == 0)
                                mode = LOCK_READ;
                        else
                                mode = LOCK_WRITE;
                        key = atoi(clp->args[5]);
```

```
				wait = atoi(clp->args[6]);
				switch (attr_lockmode(clp, clp->args[2],
				clp->args[3], mode, &key)) {
				 case ATTR_SUCCESS:
					(void) sprintf(buf, "%ld is key", key);
					reply(clp->fd, RS_ACCEPT, RT_DM, RG_OK
					reply_end(clp->fd);
					break;
				 case ATTR_LOCKWAIT:
					if (wait)
						attr_sleep(clp, clp->args[2],
							clp->args[3], LOCK_UPG
					else {
						reply(clp->fd, RS_FAILED, RT_DM,
							RB_CANTLOCK,
							"Lock unavailable");
						reply_end(clp->fd);
					}
					break;
				 case TBL_NOSUCH:
					reply(clp->fd, RS_FAILED, RT_DM, RB_NOS
						"No such table");
					reply_end(clp->fd);
					break;
				 case ATTR_NOSUCH:
					reply(clp->fd, RS_FAILED, RT_DM, RB_NOS
						"No such attribute");
					reply_end(clp->fd);
					break;
				 case ATTR_NOPERM:
					reply(clp->fd, RS_FAILED, RT_DM, RB_BAD
						"Key does not match lock");
					reply_end(clp->fd);
					break;
				 default:
					error_internal("Lockmode attribute failed");
					/* NOTREACHED */
				}
			}
			else
				c_syntax(clp);
		}
		else
			c_badcount(clp);
		return 0;
}

/*
 * c_unlock:
 *	Execute UNLOCK command
 */
static
int
c_unlock(clp)
CLIENT	*clp;
{
	long	key;
#ifdef STATS
	extern int		nlock_com;

	nlock_com++;
#endif
	if (clp->nargs == 4) {
		if (strcmp(clp->args[1], "TABLE") == 0) {
			key = atoi(clp->args[3]);
			switch (tbl_unlock(clp, clp->args[2], key)) {
			 case TBL_SUCCESS:
				reply(clp->fd, RS_ACCEPT, RT_DM, RG_OKAY,
					"Table unlocked");
				break;
			 case TBL_NOPERM:
				reply(clp->fd, RS_FAILED, RT_DM, RB_BADKEY,
					"Key does not match lock");
				break;
			 case TBL_NOSUCH:
				reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
					"No such table");
				break;
			 default:
				error_internal("Unlock table failed");
				/* NOTREACHED */
			}
			reply_end(clp->fd);
		}
		else
			c_syntax(clp);
	}
	else if (clp->nargs == 5) {
		if (strcmp(clp->args[1], "ATTRIBUTE") == 0) {
			key = atoi(clp->args[4]);
			switch (attr_unlock(clp, clp->args[2],
			clp->args[3], key)) {
			 case ATTR_SUCCESS:
				reply(clp->fd, RS_ACCEPT, RT_DM, RG_OKAY,
```

```
						"Attribute unlocked");
					break;
				case TBL_NOSUCH:
					reply(clp->fd, RS_FAILED, RT_DM, RB_NOS
						"No such table");
					break;
				case ATTR_NOSUCH:
					reply(clp->fd, RS_FAILED, RT_DM, RB_NOS
						"No such attribute");
					break;
				case ATTR_NOPERM:
					reply(clp->fd, RS_FAILED, RT_DM, RB_BAD
						"Key does not match lock");
					break;
				default:
					error_internal("Unlock attribute failed");
					/* NOTREACHED */
				}
				reply_end(clp->fd);
			}
			else
				c_syntax(clp);
		}
		else
			c_badcount(clp);
		return 0;
}

/*
 * c_count:
 *	Execute COUNT command
 */
static
int
c_count(clp)
CLIENT	*clp;
{
	register int		count;
	char			buf[BUFLEN];

	if (clp->nargs != 2 && clp->nargs != 4)
		c_badcount(clp);
	else if (strcmp(clp->args[1], "TABLE") != 0)
		c_syntax(clp);
	else if (clp->nargs == 4 && strcmp(clp->args[2], "CLASS") != 0)
		c_syntax(clp);
	else {
		if (clp->nargs == 4)
			count = tbl_count(TRUE, atoi(clp->args[3]));
		else
			count = tbl_count(FALSE, 0);
		(void) sprintf(buf, "%d tables", count);
		reply(clp->fd, RS_ACCEPT, RT_DM, RG_OKAY, buf);
		reply_end(clp->fd);
	}
	return 0;
}

/*
 * c_list:
 *	Execute LIST command
 */
static
int
c_list(clp)
CLIENT	*clp;
{
	register int		n;

	if (clp->nargs == 2) {
		if (strcmp(clp->args[1], "TABLE") == 0) {
			reply(clp->fd, RS_ACCEPT, RT_DM, RG_OKAY,
				"List of tables follows");
			reply_begin_ASCII(clp->fd);
			tbl_list(clp, FALSE, 0);
			reply_end_ASCII(clp->fd);
			reply_end(clp->fd);
		}
		else
			c_syntax(clp);
	}
	else if (clp->nargs == 4) {
		if (strcmp(clp->args[1], "TABLE") == 0) {
			if (strcmp(clp->args[2], "CLASS") == 0) {
				reply(clp->fd, RS_ACCEPT, RT_DM, RG_OKAY,
					"List of tables follows");
				reply_begin_ASCII(clp->fd);
				tbl_list(clp, TRUE, atoi(clp->args[3]));
				reply_end_ASCII(clp->fd);
				reply_end(clp->fd);
			}
			else
				c_syntax(clp);
		}
```

```
		else if (strcmp(clp->args[1], "ATTRIBUTE") == 0) {
			n = atoi(clp->args[3]);
			switch (attr_list(clp, clp->args[2], n)) {
			 case ATTR_SUCCESS:
				break;
			 case TBL_NOSUCH:
				reply(clp->fd, RS_FAILED, RT_DM, RB_NOS
					"No such table");
				break;
			 case TBL_NOPERM:
				reply_noperm(clp->fd);
				break;
			 default:
				error_internal("list attribute failed");
				/* NOTREACHED */
			}
			reply_end(clp->fd);
		}
		else
			c_syntax(clp);
	}
	else
		c_badcount(clp);
	return 0;
}

/*
 * c_empty:
 *	Execute EMPTY command
 */
static
int
c_empty(clp)
CLIENT	*clp;
{
	int	from, to;

	if (clp->nargs != 5) {
		c_badcount(clp);
		return 0;
	}
	from = atoi(clp->args[3]);
	to = atoi(clp->args[4]);
	switch (cell_empty(clp, clp->args[1], clp->args[2], from, to)) {
	 case TBL_NOSUCH:
		reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH, "No such table
		break;
	 case ATTR_NOSUCH:
		reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
			"No such attribute");
		break;
	 case ATTR_NOPERM:
		reply_noperm(clp->fd);
		break;
	 case INST_SUCCESS:
		reply(clp->fd, RS_ACCEPT, RT_DM, RG_OKAY, "Data deleted");
		break;
	 default:
		error_internal("Empty data failed");
		/* NOTREACHED */
	}
	reply_end(clp->fd);
	return 0;
}

/*
 * c_describe:
 *	Execute DESCRIBE command
 */
static
int
c_describe(clp)
CLIENT	*clp;
{
	if (clp->nargs == 2) {
		if (strcmp(clp->args[1], "DATASET") == 0) {
			ds_describe(clp->fd);
			reply_end(clp->fd);
		}
		else
			c_syntax(clp);
	}
	else if (clp->nargs == 3) {
		if (strcmp(clp->args[1], "TABLE") == 0) {
			switch (tbl_describe(clp, clp->args[2])) {
			 case TBL_SUCCESS:
				break;
			 case TBL_NOSUCH:
				reply(clp->fd, RS_FAILED, RT_DM, RB_NOSUCH,
					"No such table");
				break;
			 case TBL_NOPERM:
				reply_noperm(clp->fd);
				break;
```

```
			  default:
				error_internal("Describe table failed");
				/* NOTREACHED */
			}
			reply_end(clp->fd);
		}
		else
			c_syntax(clp);
	}
	else if (clp->nargs == 4) {
		if (strcmp(clp->args[1], "ATTRIBUTE") == 0) {
			switch (attr_describe(clp, clp->args[2],
			clp->args[3])) {
			  case ATTR_SUCCESS:
				break;
			  case TBL_NOSUCH:
				reply(clp->fd, RS_FAILED, RT_DM, RB_NOS
					"No such table");
				break;
			  case ATTR_NOSUCH:
				reply(clp->fd, RS_FAILED, RT_DM, RB_NOS
					"No such attribute");
				break;
			  case ATTR_NOPERM:
				reply_noperm(clp->fd);
				break;
			  default:
				error_internal("Describe attribute failed");
				/* NOTREACHED */
			}
			reply_end(clp->fd);
		}
		else
			c_syntax(clp);
	}
	else
		c_badcount(clp);
	return 0;
}
```

dm/reply.c

```
/*
 * $Header: reply.c,v 1.5 88/09/07 14:52:54 conrad Exp $
 * $Log:        reply.c,v $
 * Revision 1.5  88/09/07  14:52:54  conrad
 * Buffer output until reply_end() is called
 *
 * Revision 1.4  88/08/08  09:46:41  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.3  88/08/04  16:26:32  conrad
 * Support data type STRING and TABLE
 *
 * Revision 1.2  88/07/27  20:21:40  conrad
 * Use reply parameter instead of hardwired numbers
 *
 * Revision 1.1  88/07/25  13:15:30  conrad
 * Initial revision
 *
 */

#include "reply.h"
#include "error.h"
#include "datatype.h"
#include "table.h"

/*
 * The reply buffer is at most the length of the longest string plus
 * the status codes plus a space for separation plus a newline plus
 * a null.
 */
/*
 * reply:
 *      Send a reply to a file descriptor
 */
void
reply(fd, status, class, detail, string)
int     fd;
int     status, class, detail;
char    *string;
{
        register int        len;
        char                reply_buf[BUFLEN];

        if (status < 0 || status > 15) {
                error_internal("Reply status out of range");
                /* NOTREACHED */
        }
        if (class < 0 || class > 15) {
                error_internal("Reply class out of range");
                /* NOTREACHED */
        }
        if (detail < 0 || detail > 15) {
                error_internal("Reply detail out of range");
                /* NOTREACHED */
        }
        len = 3 + 1 + strlen(string) + 1;
        if (len > IO_SIZE) {
                error_internal("Reply too long");
                /* NOTREACHED */
        }
        (void) sprintf(reply_buf, "%x%x%x %s\n", status, class, detail, string);
        reply__write(fd, reply_buf, len);
}

#define REPLY_BUFSIZ    (BUFLEN * 4)
static char             reply__buf[REPLY_BUFSIZ];
static int              reply__cnt;

/*
 * reply__write:
 *      "write" a buffer.  Actually, just copy the data into a
 *      static buffer (if there is space)
 */
void
reply__write(fd, buf, len)
int     fd;
char    *buf;
int     len;
{
        struct iovec        iov[2];
        extern int          bcopy();

        if (REPLY_BUFSIZ - reply__cnt >= len) {
                (void) bcopy(buf, &reply__buf[reply__cnt], len);
                reply__cnt += len;
        }
        else {
                iov[0].iov_base = reply__buf;
                iov[0].iov_len = reply__cnt;
                iov[1].iov_base = buf;
                iov[1].iov_len = len;
                if (writev(fd, iov, 2) != reply__cnt + len) {
                        error_internal("reply__write failed");
```

```
				/* NOTREACHED */
			}
			reply__cnt = 0;
		}
}

/*
 * reply__putchar:
 *	"write" a single character
 */
void
reply__putchar(fd, c)
int	fd;
char	c;
{
	if (reply__cnt >= REPLY_BUFSIZ) {
		if (write(fd, reply__buf, REPLY_BUFSIZ) != REPLY_BUFSIZ) {
			error_internal("reply__putchar failed");
			/* NOTREACHED */
		}
		reply__cnt = 0;
	}
	reply__buf[reply__cnt++] = c;
}

/*
 * reply_end:
 *	Send end-of-reply message to file descriptor
 */
void
reply_end(fd)
int	fd;
{
	struct iovec	iov[2];
	static char	eor[] = "600 End of reply\n";

	if (reply__cnt == 0) {
		if (write(fd, eor, sizeof eor - 1) != sizeof eor - 1) {
			error_internal("Reply_end error");
			/* NOTREACHED */
		}
	}
	else {
		iov[0].iov_base = reply__buf;
		iov[0].iov_len = reply__cnt;
		iov[1].iov_base = eor;
		iov[1].iov_len = sizeof eor - 1;
		if (writev(fd, iov, 2) != reply__cnt + sizeof eor - 1) {
			error_internal("Reply_end error");
			/* NOTREACHED */
		}
		reply__cnt = 0;
	}
}

/*
 * reply_begin_ASCII:
 *	Start sending ASCII data
 */
void
reply_begin_ASCII(fd)
int	fd;
{
	static char	boa[] = "202 ASCII data follows\n";

	reply__write(fd, boa, sizeof boa - 1);
}

/*
 * reply_ASCII:
 *	Send ASCII data
 */
void
reply_ASCII(fd, s)
int	fd;
char	*s;
{
	register int	len;
	static char	newline = '\n';

	len = strlen(s);
	reply__write(fd, s, len);
	if (s[len - 1] != '\n')
		reply__putchar(fd, '\n');
}

/*
 * reply_end_ASCII:
 *	Terminate ASCII data
 */
void
reply_end_ASCII(fd)
int	fd;
{
```

```
	reply__write(fd, ".\n", 2);
}

/*
 * reply_noperm:
 *	Send permission denied message
 */
void
reply_noperm(fd)
int	fd;
{
	static char	pd[] = "402 Permission denied\n";

	reply__write(fd, pd, sizeof pd - 1);
}

/*
 * reply_data:
 *	Send data to client
 */
void
reply_data(fd, data_mode, dtype, dsize, from, len, data, ap)
int	fd;
int	data_mode;
int	dtype, dsize;
int	from, len;
char	*data;
ATTR	*ap;
{
	register int	size;
	char		buf[BUFLEN];

	(void) sprintf(buf, "Values for instances %d %d", from, from + len - 1);
	reply(fd, RS_ACCEPT, RT_DM, RG_OKAY, buf);

	/*
	 * If we are sending strings (or table names), just send
	 * each one followed by a newline
	 */
	if (dtype == DM_S_STRING || dtype == DM_S_TABLE) {
		reply_begin_ASCII(fd);
		dt_str_out(fd, len, data, ap);
		reply_end_ASCII(fd);
		return;
	}

	/*
	 * Binary mode.  Just compute the total size,
	 * tell the other end the number of bytes, and blast
	 */
	if (data_mode == REPLY_IN_BINARY) {
		size = dt_size(dtype, dsize) * len;
		(void) sprintf(buf, "%d bytes of binary data follows", size);
		reply(fd, RS_ACCEPT, RT_OTHER, 3, buf);
		reply__write(fd, data, size);
		return;
	}

	/*
	 * Here comes grunge city!  We need to figure out the type
	 * of the data to be sent.  Convert each one into ASCII and
	 * send it.
	 */
	reply_begin_ASCII(fd);
	dt_send(fd, dtype, dsize * len, data);
	reply_end_ASCII(fd);
}
```

dm/reply.h

```
/*
 * $Header: reply.h,v 1.4 88/09/07 14:52:20 conrad Exp $
 * $Log:          reply.h,v $
 * Revision 1.4  88/09/07  14:52:20  conrad
 * Declare internal functions reply__write and reply__putchar
 *
 * Revision 1.3  88/08/08  09:46:42  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.2  88/07/27  20:21:17  conrad
 * Use reply parameter instead of hardwired numbers
 *
 * Revision 1.1  88/07/25  13:15:30  conrad
 * Initial revision
 *
 */

#ifndef REPLY_INCLUDE

#include "config.h"

#define  MACHINE_TYPE   "vax"

#define  REPLY_IN_ASCII  0
#define  REPLY_IN_BINARY1

#define  REPLY_LENGTH   (BUFLEN - 5)

#define  RS_INFO        1
#define  RS_ACCEPT      2
#define  RS_MORE        3
#define  RS_FAILED      4
#define  RS_REJECT      5
#define  RS_END         6

#define  RT_OTHER       0
#define  RT_DM          1
#define  RT_CS          2
#define  RT_EXTENSION   8
#define  RT_DEBUG       9

#define  RG_OKAY        0
#define  RG_ASCII       0
#define  RG_BINARY      1
#define  RG_RW          0
#define  RG_RO          1

#define  RB_NOSUCH      0
#define  RB_EXISTS      1
#define  RB_UNAVAIL     2
#define  RB_CANTLOCK    3
#define  RB_BADKEY      4
#define  RB_DEADLOCK    5
#define  RB_UNSUPPORTED6

/*
 * Access functions
 */
void     reply();                /* Send a reply to a file descriptor */
void     reply_end();            /* Send end-of-reply message */
void     reply_begin_ASCII();    /* Begin sending ASCII data */
void     reply_ASCII();          /* Send an ASCII string */
void     reply_end_ASCII();      /* End sending ASCII data */
void     reply_noperm();         /* Send Permission Denied message */
void     reply_data();           /* Send data to client */
void     reply__write();
void     reply__putchar();

#define  REPLY_INCLUDE
#endif
```

dm/strtab.c

```c
/*
 * $Header: strtab.c,v 1.3 89/03/02 22:01:12 pett Exp $
 * $Log:         strtab.c,v $
 * Revision 1.3  89/03/02  22:01:12  pett
 * declared emalloc/erealloc to return void *
 *
 * Revision 1.2  88/09/07  14:51:20  conrad
 * Only read and write if length is > 0
 *
 * Revision 1.1  88/08/04  18:36:34  conrad
 * Initial revision
 *
 */
#include "table.h"
#include <string.h>

#define   HASHSIZE        4093
#define   STABSIZE        1024

typedef struct bucket        {
          struct bucket      *next;
          ATTR               *ap;
          int                n;
}         BUCKET;

static BUCKET      *bucket[HASHSIZE];

/*
 * str_lookup:
 *        Look up a string for an attribute
 */
int
str_lookup(ap, s)
ATTR     *ap;
char     *s;
{
          register BUCKET   *bp;
          register int       hashnum;
          int                hash();
          int                str_add();
          extern void        *emalloc();

          /*
           * See if the string already exists
           */
          hashnum = hash(ap, s);
          for (bp = bucket[hashnum]; bp != NULL; bp = bp->next)
                    if (bp->ap == ap && strcmp(STR_VALUE(ap,bp->n), s) == 0)
                              return bp->n;

          /*
           * So the string is not in the string table.  We
           * have to allocate the space for the hash bucket
           * and enter the string into the appropriate string
           * table.
           */
          bp = (BUCKET *) emalloc(sizeof (BUCKET));
          bp->ap = ap;
          bp->n = str_add(ap, s);
          bp->next = bucket[hashnum];
          bucket[hashnum] = bp;

          return bp->n;
}

/*
 * str_init:
 *        Initialize a string table structure
 */
void
str_init(ap)
ATTR     *ap;
{
          ap->strtab.stab = NULL;
          ap->strtab.maxlen = ap->strtab.curlen = 0;
          ap->dsize = 1;                    /* Must be true */
}

/*
 * str_add:
 *        Add a string to a string table
 */
static
int
str_add(ap, s)
ATTR     *ap;
char     *s;
{
          register int        len, n;
          register STRTAB     *stp;
          extern void         *erealloc();

          len = strlen(s) + 1;
          stp = &ap->strtab;
```

```c
	if (stp->maxlen - stp->curlen < len) {
		stp->maxlen += STABSIZE;
		stp->stab = (char *) erealloc(stp->stab, stp->maxlen);
	}
	(void) strcpy(stp->stab + stp->curlen, s);
	n = stp->curlen;
	stp->curlen += len;
	return n;
}

/*
 * str_write:
 *	Write the string table into a file
 */
void
str_write(fp, ap)
FILE	*fp;
ATTR	*ap;
{
	(void) fwrite((char *) &ap->strtab.curlen, sizeof (int), 1, fp);
	if (ap->strtab.curlen > 0)
		(void) fwrite(ap->strtab.stab, 1, ap->strtab.curlen, fp);
}

/*
 * str_read:
 *	Read the string table from a file
 */
int
str_read(fp, ap)
FILE	*fp;
ATTR	*ap;
{
	register char		*cp, *endp;
	register BUCKET	*bp;
	register int		hashnum;
	int			n;
	int			hash();
	extern void		*emalloc();

	/*
	 * Read in string table
	 */
	if (fread((char *) &n, sizeof (int), 1, fp) != 1)
		return -1;
	ap->strtab.curlen = ap->strtab.maxlen = n;
	if (n == 0)
		return 0;
	ap->strtab.stab = (char *) emalloc(ap->strtab.curlen);
	if (fread((char *) ap->strtab.stab, 1, n, fp) != n) {
		(void) free(ap->strtab.stab);
		return -1;
	}

	/*
	 * Enter all strings into hash table
	 */
	endp = ap->strtab.stab + n;
	cp = ap->strtab.stab;
	while (cp < endp) {
		bp = (BUCKET *) emalloc(sizeof (BUCKET));
		bp->ap = ap;
		bp->n = cp - ap->strtab.stab;
		hashnum = hash(ap, cp);
		bp->next = bucket[hashnum];
		bucket[hashnum] = bp;
		while (*cp++ != '\0')			/* Skip over this string */
			continue;
	}

	return 0;
}

/*
 * hash:
 *	Hash an attribute and string into a bucket number
 */
static
int
hash(ap, s)
ATTR	*ap;
char	*s;
{
	register unsigned int	cnt;

	cnt = (int) ap;
	while (*s != '\0')
		cnt = (cnt << 3) + *s++;
	return cnt % HASHSIZE;
}
```

dm/table.c

```
/*
 * $Header: table.c,v 1.8 89/03/02 22:00:40 pett Exp $
 * $Log:          table.c,v $
 * Revision 1.8  89/03/02  22:00:40  pett
 * declared emalloc/erealloc to return void *
 *
 * Revision 1.7  88/09/28  10:43:26  conrad
 * Implement multiple groups if NGROUPS is defined
 *
 * Revision 1.6  88/09/14  14:28:46  conrad
 * Change variable "table" to "table_list" so dbx doesn't choke
 *
 * Revision 1.5  88/08/08  09:46:43  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.4  88/07/29  15:01:35  conrad
 * Check lock mode correctly
 *
 * Revision 1.3  88/07/27  20:22:20  conrad
 * Use reply parameter instead of hardwired numbers
 *
 * Revision 1.2  88/07/27  15:23:54  conrad
 * Modify output of DESCRIBE TABLE command
 *
 * Revision 1.1  88/07/25  13:15:31  conrad
 * Initial revision
 *
 */

#include "table.h"
#include "dataset.h"
#include "reply.h"
#include "error.h"
#include <ctype.h>
#include <string.h>
#include <pwd.h>

TABLE	*table_list = NULL;			/* List of tables */

/*
 * tbl_new:
 *	Create a table
 */
int
tbl_new(clp, name, class)
CLIENT	*clp;
char	*name;
int	class;
{
	register TABLE	*tp;
	struct passwd	*pp;
	char		*copy_string();
	char		*make_filename();
	extern void	*emalloc();
	extern time_t	time();

	if ((ds_permission(clp) & DS_PERM_WRITE) == 0)
		return TBL_NOPERM;

	/*
	 * See if this table already exists
	 */
	tp = tbl_find(name);
	if (tp != NULL && !tp->deleted)
		return TBL_EXISTS;

	/*
	 * If we did not find a deleted table, then create one
	 * and set up the appropriate fields.
	 */
	if (tp == NULL) {
		tp = (TABLE *) emalloc(sizeof (TABLE));
		tp->next = table_list;
		table_list = tp;
		tp->name = copy_string(name);
		tp->filename = make_filename(name);
	}

	/*
	 * None of the above.  Create the actual table
	 */
	tp->class = class;
	tp->attr = NULL;
	tp->lock = lock_find(LOCK_TABLE, (void *) tp);
	tp->uid = clp->uid;
#ifdef NGROUPS
	tp->gid = clp->groups[0];		/* This is not used anywhere */
#else
	tp->gid = clp->gid;
#endif
	pp = getpwuid((int) tp->uid);
	if (pp == NULL)
		tp->owner = "nobody";
```

```
	else
		tp->owner = copy_string(pp->pw_name);
	tp->deleted = FALSE;
	tp->modified = TRUE;
	tp->holder = NULL;
	tp->mtime = time((time_t *) 0);
	tp->atime = tp->mtime;
	tp->nattr = 0;
	ds_set_times((time_t) 0, tp->mtime);
	return TBL_SUCCESS;
}

/*
 * make_filename:
 *	Associate a filename with a table
 */
static
char *
make_filename(name)
char	*name;
{
	register char		*cp;
	char			save;
	char			*copy_string();

	for (cp = name; *cp != '\0'; cp++)
		if (!isalnum(*cp) && *cp != '_')
			break;
	if (*cp != '\0') {
		save = *cp;
		*cp = '\0';
		name = copy_string(name);
		*cp = save;
	}
	else
		name = copy_string(name);
	return name;
}

/*
 * tbl_delete:
 *	Delete a table
 */
int
tbl_delete(clp, name)
CLIENT	*clp;
char	*name;
{
	register TABLE		*tp;
	register ATTR		*ap;
	void			free_tholder();
	extern time_t		time();

	if ((ds_permission(clp) & DS_PERM_WRITE) == 0)
		return TBL_NOPERM;
	tp = tbl_find(name);
	if (tp == NULL || tp->deleted)
		return TBL_NOSUCH;
	if (lock_mode(tp->lock, clp) != LOCK_WRITE)
		return TBL_NOPERM;
	for (ap = tp->attr; ap != NULL; ap = ap->next)
		if (lock_by_other(ap->lock, clp) < 0)
			return ATTR_NOPERM;

	tp->deleted = TRUE;
	attr_release(tp->attr);
	free_tholder(tp->holder);
	tp->holder = NULL;
	(void) lock_free(tp->lock);
	tp->lock = NULL;
	ds_set_times((time_t) 0, time((time_t *) NULL));
	return TBL_SUCCESS;
}

/*
 * free_tholder:
 *	Release the table holder structure
 */
static
void
free_tholder(thp)
register THOLDER		*thp;
{
	register THOLDER		*next;

	while (thp != NULL) {
		next = thp->next;
		(void) free((char *) thp);
		thp = next;
	}
}

/*
 * tbl_open:
```

```
 *      Open a table
 */
int
tbl_open(clp, name)
CLIENT  *clp;
char    *name;
{
        register TABLE          *tp;
        register THOLDER        *thp;
        extern void             *emalloc();

        /*
         * Find the table first
         */
        tp = tbl_find(name);
        if (tp == NULL || tp->deleted)
                return TBL_NOSUCH;

        /*
         * See if client already has table open.  If so, just
         * increment the count.  Otherwise, create holder structure
         * and set count to 1
         */
        for (thp = tp->holder; thp != NULL; thp = thp->next)
                if (thp->client == clp)
                        break;
        if (thp == NULL) {
                thp = (THOLDER *) emalloc(sizeof (THOLDER));
                thp->next = tp->holder;
                tp->holder = thp;
                thp->client = clp;
                thp->count = 1;
        }
        else
                thp->count++;
        return TBL_SUCCESS;
}

/*
 * tbl_close:
 *      Close a table
 */
int
tbl_close(clp, name)
CLIENT  *clp;
char    *name;
{
        register TABLE          *tp;
        register THOLDER        *thp, *prev;
        extern void             *emalloc();

        /*
         * Find the table first
         */
        tp = tbl_find(name);
        if (tp == NULL || tp->deleted)
                return TBL_NOSUCH;

        /*
         * See if client really has table open.
         */
        prev = NULL;
        for (thp = tp->holder; thp != NULL; thp = thp->next) {
                if (thp->client == clp)
                        break;
                prev = thp;
        }
        if (thp == NULL)
                return TBL_NOPERM;
        if (--thp->count == 0) {
                if (prev == NULL)
                        tp->holder = thp->next;
                else
                        prev->next = thp->next;
                (void) free((char *) thp);
        }
        return TBL_SUCCESS;
}

/*
 * tbl_count:
 *      Return the number of tables
 */
int
tbl_count(use_class, class)
int     use_class;
int     class;
{
        register TABLE  *tp;
        register int    count;
        extern time_t   time();

        count = 0;
        for (tp = table_list; tp != NULL; tp = tp->next) {
```

```
			if (tp->deleted)
				continue;
			if (use_class && tp->class != class)
				continue;
			count++;
		}
		ds_set_times(time((time_t *) 0), (time_t) 0);
		return count;
}

/*
 * tbl_list:
 *		List the table names for the given client
 */
void
tbl_list(clp, use_class, class)
CLIENT	*clp;
int		use_class;
int		class;
{
		register TABLE	*tp;
		extern time_t		time();

		for (tp = table_list; tp != NULL; tp = tp->next) {
			if (tp->deleted)
				continue;
			if (use_class && tp->class != class)
				continue;
			reply_ASCII(CL_FILENO(clp), tp->name);
		}
		ds_set_times(time((time_t *) 0), (time_t) 0);
}

/*
 * tbl_lock:
 *		Lock a table
 */
int
tbl_lock(clp, name, mode, key)
CLIENT	*clp;
char		*name;
int		mode;
long		*key;
{
		register TABLE	*tp;

		tp = tbl_find(name);
		if (tp == NULL || tp->deleted)
			return TBL_NOSUCH;
		if (mode == LOCK_WRITE
		&& (ds_permission(clp) & DS_PERM_WRITE) == 0)
			return TBL_NOPERM;
		if ((*key = lock_get(tp->lock, mode, *key, clp)) < 0)
			return TBL_LOCKWAIT;
		return TBL_SUCCESS;
}

/*
 * tbl_lockmode:
 *		Change the mode of a lock
 */
int
tbl_lockmode(clp, name, mode, key)
CLIENT	*clp;
char		*name;
int		mode;
long		*key;
{
		register TABLE	*tp;

		tp = tbl_find(name);
		if (tp == NULL || tp->deleted)
			return TBL_NOSUCH;
		switch (lock_chmod(tp->lock, mode, key, clp)) {
		  case -1:
			return TBL_NOPERM;
		  case 0:
			return TBL_SUCCESS;
		  case 1:
			return TBL_LOCKWAIT;
		  default:
			error_internal("tbl_lockmode failed");
			/* NOTREACHED */
		}
}

/*
 * tbl_unlock:
 *		Unlock a table
 */
int
tbl_unlock(clp, name, key)
CLIENT	*clp;
char		*name;
```

```
long	key;
{
	register TABLE	*tp;

	tp = tbl_find(name);
	if (tp == NULL || tp->deleted)
		return TBL_NOSUCH;
	if (lock_release(tp->lock, key, clp) < 0)
		return TBL_NOPERM;
	return TBL_SUCCESS;
}

/*
 * tbl_sleep:
 *	Go to sleep waiting for the lock of this table
 */
void
tbl_sleep(clp, name, mode)
CLIENT	*clp;
char	*name;
int	mode;
{
	register TABLE	*tp;

	tp = tbl_find(name);
	if (tp != NULL && !tp->deleted)
		cl_sleep(tp->lock, mode, clp);
}

/*
 * tbl_describe:
 *	Describe a table
 */
int
tbl_describe(clp, tname)
CLIENT	*clp;
char	*tname;
{
	register TABLE	*tp;
	register ATTR	*ap;
	register int	n, m;
	char		buf[BUFLEN];

	/*
	 * Find the table
	 */
	tp = tbl_find(tname);
	if (tp == NULL || tp->deleted)
		return TBL_NOSUCH;

	/*
	 * Check the permission
	 */
	if ((ds_permission(clp) & DS_PERM_READ) == 0)
		return TBL_NOPERM;

	/*
	 * Send the descriptions
	 */
	reply(CL_FILENO(clp), RS_ACCEPT, RT_DM, RG_OKAY, "Description follows");
	reply_begin_ASCII(CL_FILENO(clp));

	/* Send CLASS keyword */
	(void) sprintf(buf, "class %d", tp->class);
	reply_ASCII(CL_FILENO(clp), buf);
	/* Send ATTRIBUTE keyword */
	m = n = 0;
	for (ap = tp->attr; ap != NULL; ap = ap->next) {
		if (ap->ndata > 0)
			n++;
		if (ap->ndata == ap->ncell + 1)
			m++;
	}
	(void) sprintf(buf, "attributes %d %d %d", tp->nattr, n, m);
	reply_ASCII(CL_FILENO(clp), buf);
	/* Send INSTANCE keyword */
	(void) sprintf(buf, "instances %d", inst_max(tp));
	reply_ASCII(CL_FILENO(clp), buf);
	/* Send OWNER keyword */
	(void) sprintf(buf, "owner %s", tp->owner);
	reply_ASCII(CL_FILENO(clp), buf);
	/* Send TIME keyword */
	(void) sprintf(buf, "time %ld %ld", tp->mtime, tp->atime);
	reply_ASCII(CL_FILENO(clp), buf);

	reply_end_ASCII(CL_FILENO(clp));

	/*
	 * Now update the access time of the table
	 */
	tp->atime = time((time_t *) 0);
	return TBL_SUCCESS;
}
```

dm/table.c

```
/*
 * tbl_find:
 *	Find a table by name
 */
TABLE *
tbl_find(name)
char	*name;
{
	register TABLE	*tp;

	for (tp = table_list; tp != NULL; tp = tp->next)
		if (strcmp(tp->name, name) == 0)
			return tp;
	return NULL;
}
```

dm/table.h

```c
/*
 * $Header: table.h,v 1.5 89/02/23 17:54:31 pett Exp $
 * $Log:        table.h,v $
 * Revision 1.5  89/02/23  17:54:31  pett
 * changed declarations of uids/gids to UIDTYPE/GIDTYPE
 *
 * Revision 1.4  88/09/14  14:28:59  conrad
 * Change variable "table" to "table_list" so dbx doesn't choke
 *
 * Revision 1.3  88/08/08  09:46:45  conrad
 * Standardize buffer sizes
 * Add "DESCRIBE DATASET" command
 *
 * Revision 1.2  88/08/04  16:26:33  conrad
 * Support data type STRING and TABLE
 *
 * Revision 1.1  88/07/25  13:15:32  conrad
 * Initial revision
 *
 */

#ifndef TABLE_INCLUDE

#include "config.h"
#include "client.h"
#include "lock.h"
#include "dm_server.h"

/*
 * Types of locks
 */
#define LOCK_TABLE      1
#define LOCK_ATTR       2

/*
 * Return values
 */
#define TBL_SUCCESS         0
#define TBL_NOSUCH          -1
#define TBL_EXISTS          -2
#define TBL_NOPERM          -3
#define TBL_LOCKWAIT        -4
#define ATTR_SUCCESS        0
#define ATTR_NOSUCH         -5
#define ATTR_EXISTS         -6
#define ATTR_NOPERM         -7
#define ATTR_LOCKWAIT       -8
#define ATTR_UNSUPPORTED    -9
#define INST_SUCCESS        0
#define INST_NOSUCH         -10
#define CELL_SUCCESS        0

/*
 * Data block sizes
 */
#define DE_SIZE         1024        /* Max size of chunk of data */
#define DE_FRAGMENT     128         /* Size of a fragment of data */

/*
 * Bit mask type for determining existence of data
 */
typedef unsigned char       *DEXIST;

/*
 * Linked list of data blocks
 */
typedef struct dblock       {
        struct dblock       *next;
        int                 start;      /* Starting index */
        short               size;       /* Length */
        DEXIST              exist;
        void                *data;
}       DBLOCK;

/*
 * String table structure
 */
typedef struct strtab       {
        char                *stab;
        int                 maxlen;
        int                 curlen;
}       STRTAB;

/*
 * Forward declaration (recursive data structure)
 */
typedef struct attr         ATTR;

/*
 * Table holder (clients who have the table open
 */
typedef struct tholder      {
        struct tholder      *next;
        CLIENT              *client;
```

```
        int             count;          /* Number of times table was opene */
}       THOLDER;

/*
 * Table data structure
 */
typedef struct table {
        struct table    *next;          /* Transient stuff */
        LOCK            *lock;
        char            deleted;
        char            modified;
        THOLDER         *holder;
        char            *filename;
        char            *owner;

        char            *name;          /* Permanent stuff */
        int             class;
        ATTR            *attr;
        UIDTYPE         uid;
        GIDTYPE         gid;
        time_t          mtime, atime;
        int             nattr;
}       TABLE;

/*
 * Attribute data structure
 */
struct attr {
        ATTR            *next;          /* Transient stuff */
        LOCK            *lock;
        TABLE           *table;
        DBLOCK          *dblock;
        STRTAB          strtab;
        char            modified;
        char            *owner;

        char            *name;          /* Permanent stuff */
        int             ncell;
        int             ndata;
        UIDTYPE         uid;
        GIDTYPE         gid;
        time_t          mtime, atime;
        short           dtype, dsize;
};

/*
 * Package variables
 */
extern TABLE    *table_list;

/*
 * Package macros
 */
#define STR_VALUE(ap,n) (&(ap)->strtab.stab[n])

/*
 * List of table access functions
 */
int     tbl_new();              /* Create new table */
int     tbl_delete();           /* Mark table for deletion */
int     tbl_open();             /* Open table */
int     tbl_close();            /* Close table */
int     tbl_count();            /* Count number of tables (by class) */
void    tbl_list();             /* List tables (by class) */
int     tbl_lock();             /* Lock table */
int     tbl_lockmode();         /* Check table lock mode */
int     tbl_unlock();           /* Release lock on table */
void    tbl_sleep();            /* Mark client as waiting for lock */
int     tbl_read();             /* Read table in from file */
void    tbl_flush();            /* Write table into file */
int     tbl_describe();         /* Describe a table to a client */
TABLE   *tbl_find();            /* Find a table by name */

void    attr_release();         /* Release a list of attributes */
void    attr_free();            /* Free up an attribute */
int     attr_new();             /* Create new attribute */
int     attr_delete();          /* Delete attribute */
int     attr_list();            /* List attributes in a table */
int     attr_lock();            /* Lock attribute */
int     attr_lockmode();        /* Check attribute lock mode */
int     attr_unlock();          /* Release lock on attribute */
void    attr_sleep();           /* Mark client as waiting for lock */
int     attr_describe();        /* Describe an attribute to a client */

DBLOCK  *dblock_alloc();        /* Allocate a data block */
void    dblock_release();       /* Release a list of data blocks */
void    dblock_free();          /* Free up a data block */
void    dblock_add();           /* Add a new row */

int     inst_new();             /* Add a new instance */
int     inst_delete();          /* Delete a new instance */
int     inst_max();             /* Compute maximum instance number */

int     cell_empty();           /* Clear out some data */
```

dm/table.h

```
int     str_lookup();           /* Look up a string for an attribute */
void    str_init();             /* Initialize string table */
void    str_write();            /* Write string table to file */
int     str_read();             /* Read string table from file */

#define TABLE_INCLUDE
#endif
```

libofs/attribute.c

```
/*
 * $Header: attribute.c,v 1.7 89/03/01 20:30:36 pett Exp $
 * $Log:          attribute.c,v $
 * Revision 1.7  89/03/01  20:30:36  pett
 * changed various declarations to take advantage of machine-independence
 * offered by machdep.h
 *
 * Revision 1.6  88/10/05  21:51:45  conrad
 * Remember to release the cache buffer when closing an attribute
 *
 * Revision 1.5  88/10/04  15:33:30  conrad
 * Add local data cache
 *
 * Revision 1.4  88/09/26  13:06:54  conrad
 * Make ASCII mode work with arrays
 *
 * Revision 1.3  88/09/07  15:00:44  conrad
 * Add data I/O functionality
 *
 * Revision 1.2  88/08/08  09:47:23  conrad
 * Standardize buffer sizes
 * Add support for STRING and TABLE types
 *
 * Revision 1.1  88/07/29  15:58:55  conrad
 * Initial revision
 *
 */
#include "mstrings.h"
#include "dm_internal.h"
#include "response.h"
#include <ctype.h>
#include <math.h>

/*
 * attr_create:
 *        Create an attribute within a table
 */
DM_ATTR
attr_create(th, attr, stype)
DM_TABLE        th;
char            *attr;
DM_STYPE        stype;
{
        register TABLE   *tp;
        int              status;
        char             buf[BUFLEN];

        if (tbl_lock(th, DM_WRITELOCK, DEFAULT_KEY, TRUE) < 0)
                return NULL;

        tp = (TABLE *) th;
        (void) sprintf(buf, "CREATE ATTRIBUTE \"%s\" \"%s\" %d %d\n",
                tp->name, attr, stype.base_type, stype.length);
        (void) write(fileno(tp->hp->fp), buf, strlen(buf));

        status = mmwb_resp_status(tp->hp->fp);
        (void) tbl_unlock(th, DEFAULT_KEY);

        if (dm__check_reply(status) < 0)
                return NULL;
        return attr_open(th, attr);
}

/*
 * attr_open:
 *        Open an attribute (grab a read lock)
 */
DM_ATTR
attr_open(th, attr)
DM_TABLE        th;
char            *attr;
{
        register TABLE          *tp;
        register ATTR           *ap;
        int                     key;
        int                     get_attr_lock();
        int                     free_attr_lock();
        extern MALLOCTYPE       malloc();

        tp = (TABLE *) th;
        key = get_attr_lock(tp->hp->fp, tp->name, attr, "LOCK", DM_READLOCK,
                DEFAULT_KEY, TRUE);
        if (key < 0)
                return NULL;

        ap = (ATTR *) malloc(sizeof (ATTR));
        if (ap == NULL) {
                dm_errno = DM_EOUTOFMEM;
                return NULL;
        }
        ap->name = dm__copy_string(attr);
        if (ap->name == NULL) {
                (void) free((char *) ap);
                return NULL;
```

```
	}
	ap->lock = NULL;
	if (dm__lock_set(&ap->lock, DM_READLOCK, TRUE) == LOCK_ERROR) {
		(void) free_attr_lock(tp->hp->fp, tp->name, attr, DEFAULT_KEY);
		(void) free(ap->name);
		(void) free((char *) ap);
		return NULL;
	}
	ap->key = key;
	ap->status = 0;
	ap->owner = NULL;
	ap->tp = tp;
	ap->cache = NULL;
	ap->next = tp->attr;
	tp->attr = ap;

	return (DM_ATTR) ap;
}

/*
 * get_attr_lock:
 *	Get a lock or change a lock mode
 */
static
int
get_attr_lock(fp, table, attr, command, mode, key, wait)
FILE	*fp;
char	*table;
char	*attr;
char	*command;
int	mode, key, wait;
{
	char	buf[BUFLEN];
	int	n;

	(void) sprintf(buf, "%s ATTRIBUTE \"%s\" \"%s\" %s %d %d\n", command,
		table, attr, (mode == DM_WRITELOCK) ? "write" : "read",
		key, wait);
	(void) write(fileno(fp), buf, strlen(buf));
	if (dm__resp_number(fp, &n) < 0)
		return -1;
	return n;
}

/*
 * free_attr_lock:
 *	Release a lock on an attribute
 */
static
int
free_attr_lock(fp, table, attr, key)
FILE	*fp;
char	*table;
char	*attr;
int	key;
{
	char	buf[BUFLEN];

	(void) sprintf(buf, "UNLOCK ATTRIBUTE \"%s\" \"%s\" %d\n",
		table, attr, key);
	(void) write(fileno(fp), buf, strlen(buf));
	return dm__check_reply(mmwb_resp_status(fp));
}

/*
 * attr_close:
 *	Close an attribute (release locks)
 */
int
attr_close(ah)
DM_ATTR	ah;
{
	register ATTR	*target, *ap, *prev;

	target = (ATTR *) ah;
	prev = NULL;
	for (ap = target->tp->attr; ap != NULL; ap = ap->next) {
		if (ap == target)
			break;
		prev = ap;
	}
	if (ap != target) {
		dm_errno = DM_ENOSUCH;
		return -1;
	}

	/*
	 * Remove it from the table list
	 */
	if (prev == NULL)
		ap->tp->attr = ap->next;
	else
		prev->next = ap->next;
```

```c
	/*
	 * Release locks on the attribute and release data structure
	 */
	(void) free_attr_lock(ap->tp->hp->fp, ap->tp->name, ap->name, ap->key);
	attr__free(ap);
	return 0;
}

/*
 * attr__free:
 *	Release an attribute
 */
void
attr__free(ap)
ATTR	*ap;
{
	register LOCK	*lp, *next;
	void		cache_free();

	for (lp = ap->lock; lp != NULL; lp = next) {
		next = lp->next;
		(void) free((char *) lp);
	}
	cache_free(ap);
	(void) free(ap->name);
	(void) free((char *) ap);
}

/*
 * attr_delete:
 *	Delete an attribute
 */
int
attr_delete(ah)
DM_ATTRah;
{
	register ATTR	*target, *ap, *prev;
	int		key;
	int		status;
	char		buf[BUFLEN];

	target = (ATTR *) ah;
	prev = NULL;
	for (ap = target->tp->attr; ap != NULL; ap = ap->next) {
		if (ap == target)
			break;
		prev = ap;
	}
	if (ap != target) {
		dm_errno = DM_ENOSUCH;
		return -1;
	}

	/*
	 * Remove it from the table list
	 */
	if (prev == NULL)
		ap->tp->attr = ap->next;
	else
		prev->next = ap->next;

	/*
	 * Grab a write-lock on the attribute and delete it
	 */
	key = get_attr_lock(ap->tp->hp->fp, ap->tp->name, ap->name,
		"LOCK", DM_WRITELOCK, DEFAULT_KEY, TRUE);
	if (key > 0) {
		(void) sprintf(buf, "DELETE ATTRIBUTE \"%s\" \"%s\"\n",
			ap->tp->name, ap->name);
		(void) write(fileno(ap->tp->hp->fp), buf, strlen(buf));
		status = dm__check_reply(mmwb_resp_status(ap->tp->hp->fp));
	}
	else
		status = -1;
	attr__free(ap);
	return status;
}

/*
 * attr_name:
 *	Return the name of attribute
 */
char *
attr_name(ah)
DM_ATTRah;
{
	return ((ATTR *) ah)->name;
}

/*
 * attr_lock:
 *	Lock an attribute
 */
int
```

libofs/attribute.c

```
attr_lock(ah, mode, key, wait)
DM_ATTRah;
int	mode, key, wait;
{
	register ATTR	*ap;
	register TABLE	*tp;
	int		state;
	char		*com;
	int		get_attr_lock();

	ap = (ATTR *) ah;
	state = dm__lock_state(ap->lock);
	if (state >= mode)
		return ap->key;
	tp = ap->tp;
	com = (state == DM_NOLOCK) ? "LOCK" : "LOCKMODE";
	key = get_attr_lock(tp->hp->fp, tp->name, ap->name,
		com, mode, key, wait);
	if (key < 0)
		return -1;
	return dm__lock_set(&ap->lock, mode, TRUE);
}

/*
 * attr_lock_change:
 *	Change the mode of a lock
 */
int
attr_lock_change(ah, mode, key, wait)
DM_ATTRah;
int	mode, key, wait;
{
	register ATTR	*ap;
	register TABLE	*tp;

	ap = (ATTR *) ah;
	switch (dm__lock_change(ap->lock, mode)) {
	  case LOCK_OKAY:
		return 0;
	  case LOCK_NEWMODE:
		break;
	  default:
		dm_errno = DM_ENOSUCH;
		return -1;
	}
	tp = ap->tp;
	key = get_attr_lock(tp->hp->fp, tp->name, ap->name,
			"LOCKMODE", mode, key, wait);
	if (key < 0)
		return -1;
	return dm__lock_set(&ap->lock, mode, FALSE);
}

/*
 * attr_unlock:
 *	Unlock an attribute
 */
int
attr_unlock(ah, key)
DM_ATTRah;
int	key;
{
	register ATTR	*ap;
	register TABLE	*tp;

	ap = (ATTR *) ah;
	if (key != DEFAULT_KEY && key != ap->key) {
		dm_errno = DM_EBADKEY;
		return -1;
	}
	if (ap->lock->next == NULL) {
		dm_errno = DM_EUNAVAIL;
		return -1;
	}
	switch (dm__lock_release(&ap->lock)) {
	  case LOCK_OKAY:
		return 0;
	  case LOCK_RELEASE:
		tp = ap->tp;
		return free_attr_lock(tp->hp->fp, tp->name, ap->name, key);
	  case LOCK_NEWMODE:
		tp = ap->tp;
		return get_attr_lock(tp->hp->fp, tp->name, ap->name,
			"LOCKMODE", ap->lock->state, key, TRUE);
	  default:
		dm_errno = DM_ENOSUCH;
		return -1;
	}
}

/*
 * attr_stype:
 *	Return the storage type of the attribute
 */
```

libofs/attribute.c

```
DM_STYPE
attr_stype(ah)
DM_ATTR	ah;
{
	register ATTR	*ap;
	DM_STYPE	stype;
	void		get_attr_desc();

	ap = (ATTR *) ah;
	if ((ap->status & ATTR_ST_GOTDESC) == 0)
		get_attr_desc(ap);
	stype.base_type = ap->base_type;
	stype.length = ap->length;
	return stype;
}

/*
 * get_attr_desc:
 *	Get attribute description
 */
static
void
get_attr_desc(ap)
ATTR	*ap;
{
	char	buf[BUFLEN];
	int	read_attr_desc();

	(void) sprintf(buf, "DESCRIBE ATTRIBUTE \"%s\" \"%s\"\n",
		ap->tp->name, ap->name);
	(void) write(fileno(ap->tp->hp->fp), buf, strlen(buf));
	(void) dm__read_list(ap->tp->hp, read_attr_desc,
		(void *) ap, (void *) NULL);
	ap->status |= ATTR_ST_GOTDESC;
}

/*
 * read_attr_desc:
 *	Read the actual tables descriptions
 */
/* ARGSUSED */
static
int
read_attr_desc(fp, arg1, arg2)
FILE	*fp;
void	*arg1, *arg2;
{
	register ATTR	*ap;
	char		buf[BUFLEN];
	char		owner[BUFLEN];

	ap = (ATTR *) arg1;
	while (fgets(buf, sizeof buf, fp) != NULL) {
		if (buf[0] == '.' && buf[1] == '\n')
			return 0;
		if (strncmp(buf, "instances", 9) == 0)
			(void) sscanf(buf, "instances %d %d",
				&ap->last_cell, &ap->ndata);
		else if (strncmp(buf, "time", 4) == 0)
			(void) sscanf(buf, "time %ld %ld", &ap->mtime,
				&ap->atime);

		if (ap->status & ATTR_ST_GOTDESC)
			continue;
		/* These descriptions only need to be read once */
		if (strncmp(buf, "type", 4) == 0)
			(void) sscanf(buf, "type %d %d", &ap->base_type,
				&ap->length);
		else if (strncmp(buf, "owner", 5) == 0) {
			if (ap->owner == NULL) {
				(void) sscanf(buf, "owner %s", owner);
				ap->owner = dm__copy_string(owner);
			}
		}
	}
	dm_errno = DM_EPROTOCOL;
	return -1;
}

/*
 * cell_count:
 *	Return the number of cells in an attribute
 */
int
cell_count(ah, itype)
DM_ATTR	ah;
int	itype;
{
	register ATTR	*ap;
	void		get_attr_desc();

	ap = (ATTR *) ah;
	get_attr_desc(ap);
	switch (itype) {
```

```
		case DM_I_FILLED:
			return ap->ndata;
		case DM_I_MAX:
			return ap->last_cell;
		default:
			dm_errno = DM_EBADARG;
			return -1;
		}
}

/*
 * cell_read:
 *	Read in a single cell of an attribute
 */
int
cell_read(ah, n, data)
DM_ATTRah;
int	n;
void	*data;
{
	register ATTR	*ap;
	register TABLE	*tp;
	char		buf[BUFLEN];
	int		dm__data_read();
	CACHE		*cache_alloc();
	int		cache_fill();
	int		cache_copy();

	ap = (ATTR *) ah;
	tp = ap->tp;
	if ((ap->status & ATTR_ST_GOTDESC) == 0)
		get_attr_desc(ap);

	/* No need to acquire locks since we "know" we must have at
	 * least a read lock since user has provided a handle */

	if (tp->can_cache) {
		if (ap->cache == NULL)
			ap->cache = cache_alloc(ap);
		if (ap->cache != NULL) {
			if (n < ap->cache->begin || n > ap->cache->end)
				if (cache_fill(tp, ap, n) < 0)
					return -1;
			return cache_copy(ap, n, data);
		}
	}
	(void) sprintf(buf, "READ \"%s\" \"%s\" %d %d\n",
			tp->name, ap->name, n, n);
	(void) write(fileno(tp->hp->fp), buf, strlen(buf));

	return dm__data_read(ap, data);
}

/*
 * cell_write:
 *	Write out a single cell of an attribute
 */
int
cell_write(ah, n, data)
DM_ATTRah;
int	n;
void	*data;
{
	register ATTR	*ap;
	register TABLE	*tp;
	int		status, key;
	char		buf[BUFLEN];
	int		dm__data_supported();
	void		dm__data_write();

	ap = (ATTR *) ah;
	if ((ap->status & ATTR_ST_GOTDESC) == 0)
		get_attr_desc(ap);
	if (dm__data_supported(ap->base_type) < 0) {
		dm_errno = DM_EUNSUPPORTED;
		return -1;
	}
	tp = ap->tp;

	if ((key = attr_lock(ah, DM_WRITELOCK, DEFAULT_KEY, TRUE)) < 0)
		return -1;

	if (data == NULL) {
		(void) sprintf(buf, "EMPTY \"%s\" \"%s\" %d %d\n",
				tp->name, ap->name, n, n);
		(void) write(fileno(tp->hp->fp), buf, strlen(buf));
	}
	else {
		(void) sprintf(buf, "WRITE \"%s\" \"%s\" %d %d\n",
				tp->name, ap->name, n, n);
		(void) write(fileno(tp->hp->fp), buf, strlen(buf));
		if ((status = mmwb_resp_status(tp->hp->fp)) != 310) {
			(void) attr_unlock(ah, key);
			return dm__check_reply(status);
```

```
			}
			dm__data_write(ap, data);
		}

		status = mmwb_resp_status(tp->hp->fp);
		(void) attr_unlock(ah, key);
		return (status == R_END) ? 0 : -1;
}

static int		data_size[DM_S_NTYPE] = {
		1,				/* DM_S_INT1 */
		2,				/* DM_S_INT2 */
		4,				/* DM_S_INT4 */
		8,				/* DM_S_INT8 */
		4,				/* DM_S_FLOAT4 */
		8,				/* DM_S_FLOAT8 */
		16,				/* DM_S_FLOAT16 */
		1,				/* DM_S_CHAR */
		sizeof (char *),		/* DM_S_STRING */
		1,				/* DM_S_BYTE */
		sizeof (char *),		/* DM_S_TABLE */
};

/*
 * dm__data_read:
 *	Read in a single data element
 */
static
int
dm__data_read(ap, data)
ATTR	*ap;
void	*data;
{
		HANDLE		*hp;
		DM_TABLE	*thp;
		int		status;
		int		n;
		char		buf[BUFLEN];
		char		tname[BUFLEN];
		int		ascii_input();

		hp = ap->tp->hp;
		if (fgets(buf, sizeof buf, hp->fp) == NULL) {
			dm_errno = DM_EPROTOCOL;
			return -1;
		}
		if (dm__check_reply(status = atoi(buf)) < 0) {
			while (status != R_END) {
				if (fgets(buf, sizeof buf, hp->fp) == NULL) {
					dm_errno = DM_EPROTOCOL;
					return -1;
				}
				status = atoi(buf);
			}
			return -1;
		}

		/*
		 * Okay.  This is a good reply.  Data must be coming up.
		 */
		if (fgets(buf, sizeof buf, hp->fp) == NULL) {
			dm_errno = DM_EPROTOCOL;
			return -1;
		}
		status = atoi(buf);
		if (hp->data_mode == DM_BINARY
		&& ap->base_type != DM_S_STRING && ap->base_type != DM_S_TABLE) {
			if (status != R_BINARY) {
				dm_errno = DM_EPROTOCOL;
				return -1;
			}
			n = atoi(buf + 4);
			if (fread((char *) data, 1, n, hp->fp) != n) {
				dm_errno = DM_EPROTOCOL;
				return -1;
			}
		}
		else {
			if (status != R_ASCII) {
				dm_errno = DM_EPROTOCOL;
				return -1;
			}
			if (ap->base_type == DM_S_TABLE) {
				/*
				 * We treat tables specially because we get back
				 * the table "name", not a handle.  We then have
				 * to open the table and fill in user data area
				 * with the actual "handle", not "name"
				 */
				thp = (DM_TABLE *) data;
				data = (void *) tname;
			}
			if (ascii_input(hp->fp, ap->base_type, ap->length, data) < 0)
				return -1;
```

```
		if (fgets(buf, sizeof buf, hp->fp) == NULL
		|| (buf[0] != '.' || buf[1] != '\n')) {
			dm_errno = DM_EPROTOCOL;
			/* Don't return here since we might be able to
			 * resynchronize with DM if we can get to a R_END */
		}
	}

	do
		if (fgets(buf, sizeof buf, hp->fp) == NULL) {
			dm_errno = DM_EPROTOCOL;
			return -1;
		}
	while (atoi(buf) != R_END);

	/*
	 * Okay, we got the data.  If this attribute is of type TABLE,
	 * then we must still open the table and fill in the handle
	 */
	if (ap->base_type == DM_S_TABLE)
		*thp = tbl_open((DM_HANDLE) ap->tp->hp, tname);
	return 0;
}

/*
 * ascii_input:
 *	Read in data for a single element of given type and length
 */
static
int
ascii_input(fp, stype, slength, data)
FILE	*fp;
int	stype, slength;
void	*data;
{
	char	buf[BUFLEN];
	char	*cp;
	int	htob();

	while (slength-- > 0) {
		if (fgets(buf, sizeof buf, fp) == NULL) {
			dm_errno = DM_EPROTOCOL;
			return -1;
		}
		switch (stype) {
		 case DM_S_INT1:
		 case DM_S_BYTE:
		case DM_S_CHAR:
			*(char *)data = htob(buf);
			break;
		case DM_S_INT2:
			*(short *)data = htob(buf);
			break;
		case DM_S_INT4:
			*(long *)data = htob(buf);
			break;
		case DM_S_FLOAT4:
			*(float *)data = atof(buf);
			break;
		case DM_S_FLOAT8:
			*(double *)data = atof(buf);
			break;
		case DM_S_STRING:
		case DM_S_TABLE:
			if ((cp = strchr(buf, '\n')) != NULL)
				*cp = '\0';
			(void) strcpy((char *) data, buf);
			break;
		default:
			dm_errno = DM_EUNSUPPORTED;
			break;
		}
	}
	return 0;
}

/*
 * htob:
 *	Hex to binary
 */
static
int
htob(s)
register char	*s;
{
	register int	n;

	n = 0;
	while (isxdigit(*s)) {
		if (isdigit(*s))
			n = (n << 4) | (*s - '0');
		else
			n = (n << 4) | (*s - '0' + 10);
		s++;
```

```
	}
	return n;
}

/*
 * dm__data_write:
 *	Send a single cell to the DM
 */
static
void
dm__data_write(ap, data)
ATTR	*ap;
void	*data;
{
	register int	i, n;
	HANDLE		*hp;
	char		buf[BUFLEN];
	int		len;
	char		*cp;
	short		*sp;
	long		*lp;
	float		*fp;
	double		*dp;
	void		btoh();
#define	FSIZE	14
#define	DSIZE	24

	hp = ap->tp->hp;
	if (hp->data_mode == DM_BINARY
	&&  ap->base_type != DM_S_STRING && ap->base_type != DM_S_TABLE
		(void) write(fileno(hp->fp), (char *) data,
			data_size[ap->base_type] * ap->length);
		return;
	}
	switch (ap->base_type) {
	  case DM_S_INT1:
	  case DM_S_BYTE:
	  case DM_S_CHAR:
		cp = (char *) data;
		for (i = 0; i < ap->length; i++)
			btoh(buf, (unsigned long) cp[i], &len);
		break;
	  case DM_S_INT2:
		sp = (short *) data;
		for (i = 0; i < ap->length; i++)
			btoh(buf, (unsigned long) sp[i], &len);
		break;
	  case DM_S_INT4:
		lp = (long *) data;
		for (i = 0; i < ap->length; i++)
			btoh(buf, (unsigned long) lp[i], &len);
		break;
	  case DM_S_FLOAT4:
		fp = (float *) data;
		for (n = 0, i = 0; i < ap->length; n += FSIZE, i++) {
			if (n + FSIZE > BUFSIZ) {
				(void) write(fileno(hp->fp), buf, n);
				n = 0;
			}
			(void) sprintf(&buf[n], "% .6e\n", fp[i]);
		}
		len = n;
		break;
	  case DM_S_FLOAT8:
		dp = (double *) data;
		for (n = 0, i = 0; i < ap->length; n += DSIZE, i++) {
			if (n + DSIZE > BUFSIZ) {
				(void) write(fileno(hp->fp), buf, n);
				n = 0;
			}
			(void) sprintf(&buf[n], "% .16e\n", dp[i]);
		}
		len = n;
		break;
	  case DM_S_STRING:
		(void) sprintf(buf, "%s\n", (char *) data);
		len = strlen(buf);
		break;
	  case DM_S_TABLE:
		(void) sprintf(buf, "%s\n", (*(TABLE **) data)->name);
		len = strlen(buf);
		break;
	}
	if (len > 0)
		(void) write(fileno(hp->fp), buf, len);
}

/*
 * btoh:
 *	Binary to hex
 */
static
void
btoh(buf, val, len)
```

libofs/attribute.c

```
char		*buf;
unsigned long	val;
int		*len;
{
	register char	*s;
	register int	n;
	char		revbuf[10];		/* At most 10 hex per long

	s = revbuf;
	while (val != 0) {
		n = val & 0xf;
		val >>= 4;
		*s++ = n > 9 ? (n - 10 + 'A') : (n + '0');
	}
	if (s == revbuf) {
		*len = 1 + 1;			/* '0' + '\n' */
		*buf++ = '0';
	}
	else {
		*len = (s - revbuf) + 1;
		while (s > revbuf)
			*buf++ = *--s;
	}
	*buf++ = '\n';
	*buf = '\0';
}

/*
 * dm__data_supported:
 *	Is the given data type supported?
 */
static
int
dm__data_supported(stype)
int	stype;
{
	switch (stype) {
	  case DM_S_INT1:
	  case DM_S_BYTE:
	  case DM_S_CHAR:
	  case DM_S_INT2:
	  case DM_S_INT4:
	  case DM_S_FLOAT4:
	  case DM_S_FLOAT8:
	  case DM_S_STRING:
	  case DM_S_TABLE:
		return 0;
	  default:
		return -1;
	}
}

/*
 * cache_alloc:
 *	Allocate space for the data cache of an attribute
 */
static
CACHE *
cache_alloc(ap)
ATTR	*ap;
{
	register CACHE	*cp;
	extern char	*malloc();

	cp = (CACHE *) malloc(sizeof (CACHE));
	if (cp == NULL)
		return NULL;
	cp->begin = 0;
	cp->end = -1;
	cp->exist = cp->changed = 0;
	cp->data = (char *) malloc((unsigned) CACHE_SIZE *
		data_size[ap->base_type] * ap->length);
	if (cp->data == NULL) {
		(void) free((char *) cp);
		return NULL;
	}
	return cp;
}

/*
 * cache_free:
 *	Release a cache buffer
 */
static
void
cache_free(ap)
ATTR	*ap;
{
	register CACHE	*cp;
	register int	n;
	register char	**cpp;

	cp = ap->cache;
	if (cp == NULL)
```

```
			return;
		if (ap->base_type == DM_S_STRING || ap->base_type == DM_S_TABLE) {
			cpp = (char **) cp->data;
			for (n = 0; n < CACHE_SIZE; n++) {
				if ((cp->exist & (1 << n)) == 0)
					continue;
				mmwb_stringfree(cpp[n]);
			}
		}
		(void) free((char *) cp);
		ap->cache = NULL;
}

/*
 * cache_fill:
 *	Fill the cache starting with the given row
 */
static
int
cache_fill(tp, ap, row)
TABLE	*tp;
ATTR	*ap;
int	row;
{
	register CACHE	*cp;
	register int	n, esize;
	register char	**cpp;
	register int	status;
	register FILE	*fp;
	int		begin, end;
	int		failed;
	char		buf[BUFLEN];
	int		get_rows();
	int		get_ascii();

	/*
	 * Release old data if necessary
	 */
	cp = ap->cache;
	if (ap->base_type == DM_S_STRING || ap->base_type == DM_S_TABLE) {
		cpp = (char **) cp->data;
		for (n = 0; n < CACHE_SIZE; n++) {
			if ((cp->exist & (1 << n)) == 0)
				continue;
			mmwb_stringfree(cpp[n]);
		}
	}

	/*
	 * Set new cache bounds
	 */
	cp->begin = row;
	cp->end = row + CACHE_SIZE - 1;
	cp->exist = cp->changed = 0;
	fp = tp->hp->fp;
	esize = data_size[ap->base_type] * ap->length;

	/*
	 * Send request to data manager
	 */
	(void) sprintf(buf, "READ \"%s\" \"%s\" %d %d\n", tp->name, ap->name,
		cp->begin, cp->end);
	(void) write(fileno(fp), buf, strlen(buf));

	/*
	 * Loop over replies
	 */
	failed = FALSE;
	for (;;) {
		/*
		 * Check for reply that says data is coming
		 * (210 data for rows x y)
		 */
		if (fgets(buf, sizeof buf, fp) == NULL) {
			dm_errno = DM_EPROTOCOL;
			return -1;
		}
		status = atoi(buf);
		if (status == R_END)
			break;
		if (dm__check_reply(status) < 0
		|| get_rows(buf, &begin, &end) < 0) {
			while (status != R_END) {
				if (fgets(buf, sizeof buf, fp) == NULL) {
					dm_errno = DM_EPROTOCOL;
					return -1;
				}
				status = atoi(buf);
			}
			return -1;
		}

		/*
		 * Read the incoming data
```

```
		 */
		begin -= cp->begin;
		end -= cp->begin;
		if (fgets(buf, sizeof buf, fp) == NULL) {
			dm_errno = DM_EPROTOCOL;
			return -1;
		}
		status = atoi(buf);
		if (tp->hp->data_mode == DM_BINARY
		&&  ap->base_type != DM_S_STRING
		&&  ap->base_type != DM_S_TABLE) {
			if (status != R_BINARY) {
				dm_errno = DM_EPROTOCOL;
				return -1;
			}
			n = atoi(buf + 4);
			if ((end - begin + 1) * esize != n) {
				dm_errno = DM_EPROTOCOL;
				failed = TRUE;
			}
			if (fread(cp->data + (begin * esize), 1, n, fp) != n) {
				dm_errno = DM_EPROTOCOL;
				return -1;
			}
			for (n = begin; n <= end; n++)
				cp->exist |= (1 << n);
		}
		else {
			if (status != R_ASCII) {
				dm_errno = DM_EPROTOCOL;
				return -1;
			}
			if (get_ascii(fp, ap, begin, end) < 0) {
				failed = TRUE;
				break;
			}
			if (fgets(buf, sizeof buf, fp) == NULL
			||  (buf[0] != '.' || buf[1] != '\n')) {
				dm_errno = DM_EPROTOCOL;
				failed = TRUE;
			}
		}
	}

	return failed ? -1 : 0;
}

/*
 * get_rows:
 *	Get the row numbers of the answer
 */
static
int
get_rows(buf, begin, end)
char	*buf;
int	*begin, *end;
{
	register char	*cp;

	cp = strrchr(buf, ' ');
	if (cp == NULL)
		return -1;
	*cp++ = '\0';
	*end = atoi(cp);
	cp = strrchr(buf, ' ');
	if (cp == NULL)
		return -1;
	*begin = atoi(cp + 1);
	return 0;
}

/*
 * get_ascii:
 *	Get ASCII input for cache
 */
static
int
get_ascii(fp, ap, from, to)
FILE	*fp;
ATTR	*ap;
int	from, to;
{
	register int	n, len;
	register char	*cp, *np;
	char		buf[BUFLEN];
	int		htob();

	cp = ap->cache->data + from * data_size[ap->base_type] * ap->length;
	for (n = from; n <= to; n++) {
		for (len = ap->length; len > 0; len--) {
			if (fgets(buf, sizeof buf, fp) == NULL) {
				dm_errno = DM_EPROTOCOL;
				return -1;
			}
```

```
			switch (ap->base_type) {
			 case DM_S_INT1:
			 case DM_S_BYTE:
			 case DM_S_CHAR:
				*cp = htob(buf);
				break;
			 case DM_S_INT2:
				*(short *) cp = htob(buf);
				break;
			 case DM_S_INT4:
				*(long *) cp = htob(buf);
				break;
			 case DM_S_FLOAT4:
				*(float *) cp = atof(buf);
				break;
			 case DM_S_FLOAT8:
				*(double *) cp = atof(buf);
				break;
			 case DM_S_STRING:
			 case DM_S_TABLE:
				if ((np = strchr(buf, '\n')) != NULL)
					*np = '\0';
				*(char **) cp = mmwb_cpystring(buf);
				break;
			 default:
				dm_errno = DM_EUNSUPPORTED;
				return -1;
			}
			cp += data_size[ap->base_type];
		}
		ap->cache->exist |= (1 << n);
	}
	return 0;
}

/*
 * cache_copy:
 *	Copy an item from cache to user space
 */
static
int
cache_copy(ap, n, data)
ATTR	*ap;
int	n;
void	*data;
{
	register char	*cp;
	register int	esize;

	if (n < ap->cache->begin || n > ap->cache->end) {
		dm_errno = DM_EBADCACHE;
		return -1;
	}
	n -= ap->cache->begin;
	if ((ap->cache->exist & (1 << n)) == 0) {
		dm_errno = DM_ENODATA;
		return -1;
	}
	switch (ap->base_type) {
	 case DM_S_TABLE:
		cp = ((char **) ap->cache->data)[n];
		*((DM_TABLE *) data) = tbl_open((DM_HANDLE) ap->tp->hp, cp);
		break;
	 case DM_S_STRING:
		cp = ((char **) ap->cache->data)[n];
		(void) strcpy((char *) data, cp);
		break;
	 default:
		esize = data_size[ap->base_type] * ap->length;
		cp = ap->cache->data + (n * esize);
		bcopy(cp, (char *) data, esize);
		break;
	}
	return 0;
}
```

libofs/dm.h

```
/*
 * $Header: /usr/src/local/mmwb/src/platform/libofs/RCS/dm.h,v 1.11 89/03/03 19:14:1
 */
#ifndef DM_INCLUDE
#include <machdep.h>

typedef void    *DM_HANDLE;
typedef void    *DM_TABLE;
typedef void    *DM_ATTR;

typedef int     DM_KEY;
#define DM_NO_KEY       0

typedef int     DM_ATYPE;
#define DM_A_EXIST      0
#define DM_A_FILLED     1
#define DM_A_FULL       2

typedef int     DM_ITYPE;
#define DM_I_MAX        0
#define DM_I_FILLED     1

typedef int     DM_LMODE;
#define DM_NOLOCK       -1
#define DM_READLOCK     0
#define DM_WRITELOCK    1

typedef struct {
        int     base_type;
        int     length;
}               DM_STYPE;
#define DM_S_INT1       0
#define DM_S_INT2       1
#define DM_S_INT4       2
#define DM_S_INT8       3
#define DM_S_FLOAT4     4
#define DM_S_FLOAT8     5
#define DM_S_FLOAT16    6
#define DM_S_CHAR       7
#define DM_S_STRING     8
#define DM_S_BYTE       9
#define DM_S_TABLE      10
#define DM_S_NTYPE      11

#define DM_SERVICE      "dm"

extern int      dm_errno;
extern char     *dm_errlist[];
#define DM_ENOERROR     0
#define DM_ESYSERROR    1
#define DM_ETOOSMALL    2
#define DM_ENOPERM      3
#define DM_ENODATA      4
#define DM_ECANTLOCK    5
#define DM_EPROTOCOL    6
#define DM_EDISPATCHER  7
#define DM_EHANDSHAKE   8
#define DM_ENOSUCH      9
#define DM_EEXISTS      10
#define DM_EOUTOFMEM    11
#define DM_EBADKEY      12
#define DM_EDEADLOCK    13
#define DM_EUNSUPPORTED 14
#define DM_EUNAVAIL     15
#define DM_EBADARG      16
#define DM_EBADPROJ     17
#define DM_ETOOMANY     18
#define DM_EBADPATHID   19
#define DM_EBADCACHE    20

/*
 * Some constants for simplifying implementation
 */
#define DM_TBL_NAMELEN  255
#define DM_ATTR_NAMELEN 255

/*
 * Access routines
 */
extern DM_HANDLE        dm_open_path();
extern DM_HANDLE        dm_create_path();
extern int              dm_close_path();
extern int              dm_delete_path();
extern char             *dm_path_id();
extern int              dm_times();
extern char             *dm_error();
extern void             dm_sync();

extern int              tbl_count();
extern int              tbl_count_by_class();
extern int              tbl_list();
extern int              tbl_list_by_class();
extern DM_TABLE         tbl_open();
extern int              tbl_close();
```

libofs/dm.h

```
extern DM_TABLE     tbl_create();
extern DM_TABLE     tbl_raw_create();
extern int          tbl_delete();
extern int          tbl_class();
extern char         *tbl_name();
extern int          tbl_lock();
extern int          tbl_lock_change();

extern int          attr_count();
extern int          attr_list();
extern DM_ATTR      attr_create();
extern DM_ATTR      attr_open();
extern int          attr_delete();
extern char         *attr_name();
extern int          attr_lock();
extern int          attr_lock_change();
extern int          attr_unlock();
extern DM_STYPE     attr_stype();

extern int          cell_count();
extern int          cell_read();
extern int          cell_write();

#define DM_INCLUDE
#endif
```

libofs/dm_class.h

```
#ifndef DM_CL_NONE

#define DM_CL_NONE     0
#define DM_CL_OBJECT   1

#define OBJ_CL_SIMPLE    0      /* Simple object */
#define OBJ_CL_PROJECT   1      /* Project (used to organize objects) */
#define OBJ_CL_COMPOSITE        2       /* Composite object which contains
#define OBJ_CL_TOOL      3      /* Tool object (either instrument or process) */
#define OBJ_CL_STATE     4      /* Tool state object */

#define OBJ_T_NONE       0      /* Miscellaneous object */
#define OBJ_T_PROJECT    1      /* Project object */
#define OBJ_T_MOLECULE 2        /* Molecular data */
#define OBJ_T_SEQUENCE 3        /* Molecular sequence */
#define OBJ_T_ATOM       4      /* Atomic information (but not coordinates) */
#define OBJ_T_COORD      5      /* Atomic coordinates */
#define OBJ_T_BOND       6      /* bond pair list */
#define OBJ_T_SESSION    7      /* Session manager contact object */
#define OBJ_T_PROTOCOL 8        /* Process object */
#define OBJ_T_INSTRUMENT        9       /* Instrument object */
#define OBJ_T_KARMA_SURFACE     27      /* Karma surface control points obje
#define OBJ_T_REGISTER  28      /* Registration object */

#endif
```

libofs/dm_internal.h

```c
/*
 * $Header: dm_internal.h,v 1.8 88/10/26 18:58:34 gregc Exp $
 * $Log:	dm_internal.h,v $
 * Revision 1.8  88/10/26  18:58:34  gregc
 * added OBJ_IDENT macro.
 *
 * Revision 1.7  88/10/19  17:01:23  conrad
 * Use ofs.h and OFS_OBJECT rather than dm.h and DM_OBJECT
 *
 * Revision 1.6  88/10/06  12:40:52  conrad
 * Hold on to "obj_list" table handle for projects and composite objects
 * instead of reopening it each time
 *
 * Revision 1.5  88/10/04  15:33:42  conrad
 * Add local data cache
 *
 * Revision 1.4  88/09/19  17:14:51  conrad
 * Use new dispatcher library interface
 *
 * Revision 1.3  88/09/07  14:58:50  conrad
 * Parameterize reply codes, fix up function declarations
 *
 * Revision 1.2  88/08/08  09:47:47  conrad
 * Standardize buffer sizes
 * Add support for STRING and TABLE types
 *
 * Revision 1.1  88/07/29  15:59:00  conrad
 * Initial revision
 *
 */
#ifndef CONFIG_INCLUDE

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <machdep.h>
#include "dispatcher.h"
#include "dm.h"

#ifndef	TRUE
#define	TRUE	1
#define	FALSE	0
#endif

#define	R_GENOKAY	200
#define	R_OKAY		210
#define	R_ALTOKAY	211
#define	R_ASCII		202
#define	R_BINARY	203
#define	R_NOPERM	402
#define R_NOSUCH	410
#define	R_EXISTS	411
#define	R_UNAVAIL	412
#define	R_CANTLOCK	413
#define	R_BADKEY	414
#define	R_DEADLOCK	415
#define	R_UNSUPPORTED416
#define	R_END		600

#define	BUFLEN		1024

/*
 * Data exchange mode
 */
#define	DM_ASCII	0
#define	DM_BINARY	1

/*
 * "Stackable" lock structure
 */
typedef struct lock	{
	struct lock	*next;
	int		state;
}	LOCK;
#define	LOCK_ERROR	(-1)
#define	LOCK_OKAY	0
#define	LOCK_NEWMODE 1
#define	LOCK_RELEASE	2
#define	DEFAULT_KEY	0

/*
 * Data set handle
 */
typedef struct handle	{
	FILE			*fp;
	int			version;
	int			data_mode;
	time_t			atime, mtime;
	char			*name;
	DISPATCHER_SERVER	server;
}	HANDLE;
#define	DM_READWRITE	0
#define	DM_READONLY	1
```

libofs/dm_internal.h

```
/*
 * Data cache structure
 */
typedef struct cache        {
        int     begin, end;
        long    exist, changed;
        char    *data;
}       CACHE;
#define CACHE_VALID     0x1
#define CACHE_SIZE      32

/*
 * Table and attribute handles
 */
typedef struct table        TABLE;          /* Forward declaration */

typedef struct attr         {
        struct attr     *next;
        char            *name;
        TABLE           *tp;
        LOCK            *lock;
        int             key;
        int             status;
        char            *owner;
        time_t          atime, mtime;
        int             last_cell, ndata;
        int             base_type, length;
        CACHE           *cache;
}       ATTR;
#define ATTR_ST_GOTDESC 0x1

struct table        {
        char            *name;
        HANDLE          *hp;
        ATTR            *attr;
        LOCK            *lock;
        int             key;
        int             status;
        char            *owner;
        int             class;
        int             max_inst;
        time_t          atime, mtime;
        int             nattr_exist, nattr_filled, nattr_full;
        int             can_cache;
};
#define TBL_ST_GOTDESC  0x1

typedef struct object       {
        struct object   *next;
        long            last_change;
        char            *name;
        short           type, class;
        DM_HANDLE       dmh;
        DM_TABLE        table;
        DM_TABLE        obj_list;       /* Table of subobjects */
        struct object   *parent;
        int             row;
        int             ref_count;
        struct object   *objects;
        int             delete;
}       OBJECT;

/*
 * Local macros
 */
#define OBJ_NAME(oh)    (((OBJECT *) oh)->name)
#define OBJ_TYPE(oh)    (((OBJECT *) oh)->type)
#define OBJ_CLASS(oh)   (((OBJECT *) oh)->class)
#define OBJ_DMH(oh)     (((OBJECT *) oh)->dmh)
#define OBJ_TABLE(oh)   (((OBJECT *) oh)->table)
#define OBJ_PARENT(oh)  (((OBJECT *) oh)->parent)
#define OBJ_ROW(oh)     (((OBJECT *) oh)->row)
#define OBJ_REFCNT(oh)  (((OBJECT *) oh)->ref_count)
#define OBJ_OBJECTS(oh) (((OBJECT *) oh)->objects)
#define OBJ_IDENT(oh)   (((OBJECT *) oh)->row)

/*
 * Predefined constants
 */
#define PROJECT_TABLE_NAME      "project_table"
#define MMWB_ENV                "MMWB"
#define PROTOTYPE_PROJECT       "/usr/local/lib/mmwb"

/*
 * Local access functions
 */
extern int      dm__lock_state();
extern int      dm__lock_change();
extern int      dm__lock_set();
extern int      dm__lock_release();
extern char     *dm__copy_string();
extern int      dm__read_list();
extern int      dm__check_reply();
extern int      dm__resp_number();
```

libofs/dm_internal.h

```
extern void        dm__sanitize();
extern void        dm__parse_path_id();

extern void        tbl__make_name();

extern void        attr__free();

extern OBJECT      *obj__get();
extern OBJECT      *obj__make();
extern void        obj__free();

#define  CONFIG_INCLUDE
#endif
```

libofs/error.c

```
/*
 * $Header: error.c,v 1.5 88/10/14 12:14:16 gregc Exp $
 * $Log:		error.c,v $
 * Revision 1.5  88/10/14  12:14:16  gregc
 * return's dispatcher's error string when ofs error is DM_EDISPATCHER.
 *
 * Revision 1.4  88/10/13  18:57:39  gregc
 * replace dm_perror() with dm_error().
 *
 * Revision 1.3  88/10/04  15:33:43  conrad
 * Add local data cache
 *
 * Revision 1.2  88/09/07  15:00:04  conrad
 * Add new error messages
 *
 * Revision 1.1  88/07/29  15:59:00  conrad
 * Initial revision
 *
 */
#include "dm_internal.h"

char	*dm_errlist[] = {
	"no error",
	"system error",
	"buffer too small",
	"permission denied",
	"no data",
	"cannot lock immediately",
	"protocol error",
	"dispatcher error",
	"handshake error",
	"does not exist",
	"already exists",
	"out of memory",
	"key does not match lock",
	"deadlock",
	"unsupported",
	"unavailable",
	"illegal argument",
	"dataset not a project",
	"too many matches",
	"illegal path identifier",
	"attribute data cache corrupted",
};
int	dm_errno;

#define	MAX_DMERR	(sizeof dm_errlist / sizeof dm_errlist[0])

/*
 * dm_error:
 *	Return error message
 */
char *
dm_error()
{
	static char	buf[32];

	if (dm_errno >= 0 && dm_errno < MAX_DMERR) {
		if (dm_errno == DM_EDISPATCHER)
			return dispatcher_error();
		else
			return dm_errlist[dm_errno];
	}

	(void) sprintf(buf, "Error %d", dm_errno);
	return buf;
}
```

libofs/inst.c

```
/*
 * $Header: inst.c,v 1.2 88/09/23 20:46:55 conrad Exp $
 * $Log:		inst.c,v $
 * Revision 1.2  88/09/23  20:46:55  conrad
 * Save the error number if instance creation failed
 *
 * Revision 1.1  88/07/29  15:59:00  conrad
 * Initial revision
 *
 */
#include "dm_internal.h"

/*
 * inst_create:
 *		Create an instance
 */
int
inst_create(th)
DM_TABLE		th;
{
		register TABLE	*tp;
		int			key;
		int			n, val;
		char			buf[BUFLEN];

		tp = (TABLE *) th;
		if ((key = tbl_lock(th, DM_WRITELOCK, DEFAULT_KEY, TRUE)) < 0)
				return -1;

		(void) sprintf(buf, "CREATE INSTANCE \"%s\"\n", tp->name);
		(void) write(fileno(tp->hp->fp), buf, strlen(buf));
		val = dm__resp_number(tp->hp->fp, &n);
		if (val < 0)
				n = dm_errno;

		if (tbl_unlock(th, key) < 0)
				return -1;
		if (val < 0) {
				dm_errno = n;
				return -1;
		}
		return n;
}

/*
 * inst_delete:
 *		Delete an instance
 */
int
inst_delete(th, n)
DM_TABLE		th;
int			n;
{
		register TABLE	*tp;
		int			key;
		int			val;
		char			buf[BUFLEN];

		tp = (TABLE *) th;
		if ((key = tbl_lock(th, DM_WRITELOCK, DEFAULT_KEY, TRUE)) < 0)
				return -1;

		(void) sprintf(buf, "DELETE INSTANCE \"%s\" %d\n", tp->name, n);
		(void) write(fileno(tp->hp->fp), buf, strlen(buf));
		val = dm__check_reply(mmwb_resp_status(tp->hp->fp));

		if (tbl_unlock(th, key) < 0)
				return -1;
		return val;
}
```

libofs/lock.c

```c
/*
 * $Header: lock.c,v 1.3 89/03/08 02:10:50 gregc Exp $
 * $Log:	lock.c,v $
 * Revision 1.3  89/03/08  02:10:50  gregc
 * delint.
 *
 * Revision 1.2  89/03/01  20:30:34  pett
 * changed various declarations to take advantage of machine-independence
 * offered by machdep.h
 *
 * Revision 1.1  88/07/29  15:59:01  conrad
 * Initial revision
 *
 */
#include "dm_internal.h"

/*
 * lock_state:
 *	Type of lock held at current level
 */
int
dm__lock_state(lp)
register LOCK	*lp;
{
	if (lp == NULL)
		return DM_NOLOCK;
	return lp->state;
}

/*
 * dm_lock_change:
 *	See if any work needs to be done if the mode of lock on
 *	current level changes
 */
int
dm__lock_change(lp, mode)
register LOCK	*lp;
int		mode;
{
	if (lp == NULL)
		return LOCK_ERROR;
	if (lp->next == NULL)
		/* If this is the only lock, change the mode */
		return LOCK_NEWMODE;
	if (mode <= lp->next->state)
		/* If previous level had bigger lock, just leave it */
		return LOCK_OKAY;
		/* Need a bigger lock for this level */
		return LOCK_NEWMODE;
}

/*
 * lock_set:
 *	Add a new level of lock
 */
int
dm__lock_set(lpp, state, new_level)
LOCK	**lpp;
int	state;
{
	register LOCK		*lp;
	extern MALLOCTYPE	malloc();

	if (!new_level) {
		if (*lpp == NULL)
			return LOCK_ERROR;
		(*lpp)->state = state;
		return LOCK_OKAY;
	}
	if ((lp = (LOCK *) malloc(sizeof (LOCK))) == NULL)
		return LOCK_ERROR;
	lp->state = state;
	lp->next = *lpp;
	*lpp = lp;
	return LOCK_OKAY;
}

/*
 * lock_release:
 *	Release the current level of lock
 */
int
dm__lock_release(lpp)
LOCK	**lpp;
{
	register LOCK	*lp;
	register int	retval;

	if (*lpp == NULL)
		return LOCK_ERROR;
	lp = *lpp;
	*lpp = lp->next;
	if (*lpp == NULL)
		retval = LOCK_RELEASE;
```

```
	else if ((*lpp)->state == lp->state)
		retval = LOCK_OKAY;
	else
		retval = LOCK_NEWMODE;
	(void) free((char *) lp);
	return retval;
}
```

libofs/misc.c

```
/*
 * $Header: misc.c,v 1.4 89/03/01 20:30:41 pett Exp $
 * $Log:		misc.c,v $
 * Revision 1.4  89/03/01  20:30:41  pett
 * changed various declarations to take advantage of machine-independence
 * offered by machdep.h
 *
 * Revision 1.3  88/09/14  14:25:45  conrad
 * in dm__resp_number, continue to eat responses even on error up to an R_END
 *
 * Revision 1.2  88/09/07  15:01:06  conrad
 * Add function to parse project id, parameterize reply code
 *
 * Revision 1.1  88/07/29  15:59:01  conrad
 * Initial revision
 *
 */
#include "dm_internal.h"
#include "response.h"
#include <ctype.h>

/*
 * dm__copy_string:
 *	Make a copy of the given string
 */
char *
dm__copy_string(s)
char	*s;
{
	register char		*cp;
	extern MALLOCTYPE	malloc();

	if ((cp = (char *) malloc((unsigned) strlen(s) + 1)) == NULL) {
		dm_errno = DM_EOUTOFMEM;
		return NULL;
	}
	(void) strcpy(cp, s);
	return cp;
}

/*
 * dm__check_reply:
 *	Check the reply number and set errno
 */
int
dm__check_reply(status)
int	status;
{
	switch (status) {
	 case R_GENOKAY:
	 case R_OKAY:
	 case R_ALTOKAY:
		dm_errno = DM_ENOERROR;
		break;
	 case R_NOPERM:
		dm_errno = DM_ENOPERM;
		break;
	 case R_NOSUCH:
		dm_errno = DM_ENOSUCH;
		break;
	 case R_EXISTS:
		dm_errno = DM_EEXISTS;
		break;
	 case R_UNAVAIL:
		dm_errno = DM_EUNAVAIL;
		break;
	 case R_CANTLOCK:
		dm_errno = DM_ECANTLOCK;
		break;
	 case R_BADKEY:
		dm_errno = DM_EBADKEY;
		break;
	 case R_DEADLOCK:
		dm_errno = DM_EDEADLOCK;
		break;
	 case R_UNSUPPORTED:
		dm_errno = DM_EUNSUPPORTED;
		break;
	 default:
		dm_errno = DM_EPROTOCOL;
		break;
	}
	return (status / 100 == 2) ? 0 : -1;
}

/*
 * dm__read_list:
 *	Read in a list of replies and call supplied routine to
 *	deal with ASCII data
 */
int
dm__read_list(hp, func, arg1, arg2)
HANDLE *hp;
int	(*func)();
```

```
void	*arg1, *arg2;
{
	char	buf[BUFLEN];
	int	status, val;

	if (fgets(buf, sizeof buf, hp->fp) == NULL) {
		dm_errno = DM_EPROTOCOL;
		return -1;
	}
	if ((status = atoi(buf)) == R_OKAY) {
		if (fgets(buf, sizeof buf, hp->fp) == NULL) {
			dm_errno = DM_EPROTOCOL;
			return -1;
		}
		if ((status = atoi(buf)) == R_ASCII)
			val = (*func)(hp->fp, arg1, arg2);
		else {
			dm_errno = DM_EPROTOCOL;
			val = -1;
		}
	}
	else {
		dm_errno = DM_EPROTOCOL;
		val = -1;
	}
	while (status != R_END) {
		if (fgets(buf, sizeof buf, hp->fp) == NULL) {
			dm_errno = DM_EPROTOCOL;
			return -1;
		}
		status = atoi(buf);
	}
	return val;
}

/*
 * dm__resp_number:
 *	Get a response and a number
 */
int
dm__resp_number(fp, number)
FILE	*fp;
int	*number;
{
	char	buf[BUFLEN];
	int	n, status;

	if (fgets(buf, sizeof buf, fp) == NULL) {
		dm_errno = DM_EPROTOCOL;
		return -1;
	}
	if (dm__check_reply(atoi(buf)) < 0)
		status = -1;
	else {
		n = atoi(buf + 4);
		status = 0;
	}
	while (fgets(buf, sizeof buf, fp) != NULL)
		if (atoi(buf) == R_END) {
			if (status == 0)
				*number = n;
			return status;
		}
	dm_errno = DM_EPROTOCOL;
	return -1;
}

/*
 * dm__sanitize:
 *	Convert a user external name into a legal internal name
 */
void
dm__sanitize(from, to)
register char	*from, *to;
{
	while (*from != '\0') {
		if (isspace(*from)) {
			*to++ = '_';
			from++;
		}
		else if (isprint(*from))
			*to++ = *from++;
		else {
			*to++ = '#';
			from++;
		}
	}
	*to = '\0';
}

/*
 * dm__parse_path_id:
 *	Parse an "rcp" style address into account, host, and path
 */
```

libofs/misc.c

```
void
dm__parse_path_id(addr, acct, host, path)
char	*addr;
char	*acct, *host, *path;
{
	register char	*colon, *at;

	/*
	 * If addr is empty, just set the variables and return
	 */
	if (addr == NULL || *addr == '\0') {
		*acct = *host = *path = '\0';
		return;
	}

	/*
	 * Check for an account name
	 */
	at = strchr(addr, '@');
	if (at != NULL) {
		*at = '\0';
		(void) strcpy(acct, addr);
		*at++ = '@';
		addr = at;
	}
	else
		*acct = '\0';

	/*
	 * Check for host and path
	 */
	colon = strchr(addr, ':');
	if (colon != NULL) {
		*colon = '\0';
		(void) strcpy(host, addr);
		(void) strcpy(path, colon + 1);
		*colon = ':';
	}
	else {
		(void) strcpy(host, addr);
		*path = '\0';
	}
}
```

libofs/object.c

```
#include "ofs.h"
#include "dm_internal.h"

/*
 * obj__get:
 *        Get an object by its table name from a data manager
 */
OBJECT *
obj__get(dmh, tname)
DM_HANDLE      dmh;
char           *tname;
{
        register DM_TABLE        th;
        register OBJECT          *op;
        short                    class, type;
        OBJECT                   *obj__add();
        int                      get_value();

        /*
         * Get the table itself.  We don't want the information
         * in this table to be cached since we know that the input
         * will always be (attr_open, read, attr_close) of one
         * single element.
         */
        if ((th = tbl_open(dmh, tname)) == NULL)
                return NULL;
        ((TABLE *) th)->can_cache = FALSE;
        if (get_value(th, "class", (void *) &class) < 0)
                return NULL;
        if (get_value(th, "type", (void *) &type) < 0)
                return NULL;
        op = obj__add((OBJECT *) NULL, dmh, tname, class, type, -1);
        op->table = th;
        if (class == OBJ_CL_PROJECT || class == OBJ_CL_COMPOSITE) {
                if (prop_get((OFS_OBJECT) op, "obj_list",
                (void *) &op->obj_list) < 0) {
                        (void) tbl_close(th);
                        op->table = NULL;
                        dm_errno = DM_ESYSERROR;
                        return NULL;
                }
        }
        else
                op->obj_list = NULL;
        op->ref_count++;
        return op;
}

/*
 * get_value:
 *        Get the value of the cell 0 of a particular attribute
 */
static
int
get_value(th, aname, data)
DM_TABLE       th;
char           *aname;
void           *data;
{
        register DM_ATTR         ah;

        if ((ah = attr_open(th, aname)) == NULL) {
                (void) tbl_close(th);
                return -1;
        }
        if (cell_read(ah, 0, data) < 0) {
                (void) attr_close(ah);
                (void) tbl_close(th);
                return -1;
        }
        return attr_close(ah);
}

/*
 * obj__make:
 *        Create a raw object of the given type with the given table name
 */
OBJECT *
obj__make(dmh, tname, oname, oclass, otype)
DM_HANDLE      dmh;
char           *tname, *oname;
short          oclass, otype;
{
        OFS_OBJECT      oh;
        DM_TABLE        th;
        OBJECT          *op;
        char            buf[BUFLEN];
        DM_TABLE        obj__copy_prototype();
        DM_TABLE        obj__make_table();

        (void) sprintf(buf, "prototype_%d", otype);
        if ((oh = obj_open(proj__sys, buf, oclass, otype)) != NULL)
                th = obj__copy_prototype(dmh, tname, OBJ_TABLE(oh));
        else
```

```
		th = obj__make_table(dmh, tname, oclass, otype);
	if (th == NULL)
		return NULL;
	op = obj__add((OBJECT *) NULL, dmh, oname, oclass, otype, -1);
	op->ref_count++;
	op->table = th;
	return op;
}

/*
 * obj__copy_prototype:
 *	Copy a prototype object into the given data manager
 */
static
DM_TABLE
obj__copy_prototype(dmh, tname, pt)
DM_HANDLE	dmh;
char		*tname;
DM_TABLE	pt;
{
	register int		i, j, nattr, ncell;
	DM_TABLE		th;
	DM_ATTR			ah, ph;
	DM_STYPE		stype;
	char			**aname, *anamebuf;
	void			*data;
	void			obj__copy_cell();
	void			*get_buffer();
	extern MALLOCTYPE	malloc();

	th = tbl_raw_create(dmh, tname, DM_CL_OBJECT);
	if (th == NULL)
		return NULL;
	nattr = attr_count(pt, DM_A_EXIST);
	if (nattr == 0)
		return th;
	aname = (char **) malloc((unsigned) nattr * sizeof (char *));
	anamebuf = (char *) malloc((unsigned) nattr * DM_ATTR_NAMELEN);
	if (aname == NULL || anamebuf == NULL) {
		dm_errno = DM_EOUTOFMEM;
		if (aname != NULL)
			(void) free((char *) aname);
		(void) tbl_close(th);
		return NULL;
	}
	for (i = 0; i < nattr; i++)
		aname[i] = anamebuf + i * DM_ATTR_NAMELEN;
	if (attr_list(pt, DM_A_EXIST, aname, nattr) != nattr) {
		dm_errno = DM_ESYSERROR;
		(void) free((char *) aname);
		(void) free(anamebuf);
		(void) tbl_close(th);
		return NULL;
	}
	for (i = 0; i < nattr; i++) {
		ph = attr_open(pt, aname[i]);
		stype = attr_stype(ph);
		ah = attr_create(th, aname[i], stype);
		ncell = cell_count(ph, DM_I_MAX);
		data = get_buffer(stype);
		for (j = 0; j <= ncell; j++)
			obj__copy_cell(dmh, tname, aname[i],  ph, ah,
				j, stype, data);
		(void) free((char *) data);
		(void) attr_close(ph);
		(void) attr_close(ah);
	}
	return th;
}

/*
 * get_buffer:
 *	Allocate a buffer large enough for the given storage type
 */
static
void *
get_buffer(stype)
DM_STYPE	stype;
{
	unsigned int		len;
	extern MALLOCTYPE	malloc();

	switch (stype.base_type) {
	 case DM_S_INT1:
	 case DM_S_CHAR:
	 case DM_S_BYTE:
		len = stype.length;
		break;
	 case DM_S_INT2:
		len = stype.length * 2;
		break;
	 case DM_S_INT4:
	 case DM_S_FLOAT4:
		len = stype.length * 4;
```

```
		break;
	case DM_S_INT8:
	case DM_S_FLOAT8:
		len = stype.length * 8;
		break;
	case DM_S_FLOAT16:
		len = stype.length * 16;
		break;
	case DM_S_STRING:
		len = BUFLEN;
		break;
	case DM_S_TABLE:
		len = sizeof (DM_TABLE);
		break;
	}
	return (void *) malloc(len);
}

/*
 * obj__copy_cell:
 *	Copy a single cell from one attribute to another
 */
static
void
obj__copy_cell(dmh, tname, aname, from, to, row, stype, data)
DM_HANDLE	dmh;
char		*tname, *aname;
DM_ATTR		from, to;
int		row;
DM_STYPE	stype;
void		*data;
{
	char		newname[BUFLEN];
	DM_TABLE	th;

	if (cell_read(from, row, data) < 0)
		return;
	if (stype.base_type == DM_S_TABLE) {
		tbl__make_name(newname, tname, aname, row);
		th = obj__copy_prototype(dmh, newname, *((DM_TABLE *) data))
		(void) cell_write(to, row, (void *) &th);
	}
	else
		(void) cell_write(to, row, data);
}

/*
 * obj__make_table:
 *	Create a raw object of given class and type
 */
static
DM_TABLE
obj__make_table(dmh, tname, class, type)
DM_HANDLE	dmh;
char		*tname;
short		class, type;
{
	DM_TABLE	th;
	DM_ATTR		ah;
	DM_STYPE	stype;

	th = tbl_raw_create(dmh, tname, DM_CL_OBJECT);
	if (th == NULL)
		return NULL;

	stype.base_type = DM_S_INT2;
	stype.length = 1;

	ah = attr_create(th, "class", stype);
	if (ah == NULL || cell_write(ah, 0, (void *) &class) < 0) {
		(void) tbl_delete(th);
		return NULL;
	}
	(void) attr_close(ah);

	ah = attr_create(th, "type", stype);
	if (ah == NULL || cell_write(ah, 0, (void *) &type) < 0) {
		(void) tbl_delete(th);
		return NULL;
	}
	(void) attr_close(ah);
	return th;
}

/*
 * obj__add:
 *	Add an object to a list
 */
static
OBJECT *
obj__add(proj, dmh, oname, class, type, row)
OBJECT		*proj;
DM_HANDLE	dmh;
char		*oname;
```

```
short           class, type;
int             row;
{
        register OBJECT         *op;
        extern MALLOCTYPE       malloc();

        op = (OBJECT *) malloc(sizeof (OBJECT));
        if (op == NULL)
                return NULL;
        op->name = (char *) malloc((unsigned) strlen(oname) + 1);
        if (op->name == NULL) {
                (void) free((char *) op);
                return NULL;
        }
        (void) strcpy(op->name, oname);
        op->dmh = dmh;
        op->table = NULL;
        op->obj_list = NULL;
        op->ref_count = 0;
        op->delete = FALSE;
        op->objects = NULL;
        op->class = class;
        op->type = type;
        op->parent = proj;
        op->row = row;
        op->last_change = 0;
        if (proj == NULL)
                op->next = NULL;
        else {
                op->next = proj->objects;
                proj->objects = op;
        }
        return op;
}

/*
 * obj__enter:
 *      Enter a single object into a collection
 */
static
int
obj__enter(to, target)
register OBJECT *to, *target;
{
        DM_ATTRah;
        void    obj__fix_update();

        /*
         * First we update the data manager table
         */
        if (tbl_lock(to->obj_list, DM_WRITELOCK, DM_NO_KEY, TRUE) < 0)
                return -1;
        if ((target->row = inst_create(to->obj_list)) < 0)
                return -1;
        if ((ah = attr_open(to->obj_list, "class")) != NULL) {
                (void) cell_write(ah, target->row, (void *) &target->class);
                (void) attr_close(ah);
        }
        else
                return -1;
        if ((ah = attr_open(to->obj_list, "type")) != NULL) {
                (void) cell_write(ah, target->row, (void *) &target->type);
                (void) attr_close(ah);
        }
        else
                return -1;
        if ((ah = attr_open(to->obj_list, "name")) != NULL) {
                (void) cell_write(ah, target->row, (void *) target->name);
                (void) attr_close(ah);
        }
        else
                return -1;
        if ((ah = attr_open(to->obj_list, "objects")) != NULL) {
                (void) cell_write(ah, target->row, (void *) &target->table);
                (void) attr_close(ah);
        }
        else
                return -1;
        (void) tbl_unlock(to->obj_list, DM_NO_KEY);
        obj__fix_update(to);

        /*
         * Add target into object list of collection
         */
        target->next = to->objects;
        to->objects = target;
        target->parent = to;

        return 0;
}

/*
 * obj__remove:
 *      Remove a single object from a collection
```

```
 */
static
void
obj__remove(from, target)
OBJECT *from, *target;
{
	register OBJECT	*prev, *op;
	void		obj__fix_update();

	/*
	 * First we update the data manager table
	 */
	(void) inst_delete(from->obj_list, target->row);
	obj__fix_update(from);

	/*
	 * Now we remove the object from the linked list in memory
	 */
	prev = NULL;
	for (op = from->objects; op != NULL; op = op->next)
		if (op == target) {
			if (prev == NULL)
				from->objects = op->next;
			else
				prev->next = op->next;
			obj__free(op);
			return;
		}
		else
			prev = op;
}

/*
 * obj__fix_update:
 *	Fix the update attribute of the given object
 */
static
void
obj__fix_update(op)
OBJECT *op;
{
	void		obj__update();
	OFS_PROP	ph;

	obj__update(op);
	ph = prop_open((OFS_OBJECT) op, "update");
	if (prop_lock(ph, DM_READLOCK, DM_NO_KEY, TRUE) < 0) {
		(void) prop_close(ph);
		return;
	}
	op->last_change++;
	(void) prop_write(ph, (void *) &op->last_change);
	(void) prop_unlock(ph, DM_NO_KEY);
	(void) prop_close(ph);
}

/*
 * obj__free:
 *	Release an object data structure (plus all that is underneath)
 */
void
obj__free(target)
OBJECT *target;
{
	register OBJECT	*op, *next;

	if (target == NULL)
		return;
	for (op = target->objects; op != NULL; op = next) {
		next = op->next;
		obj__free(op);
	}
	if (target->name != NULL)
		(void) free(target->name);
	(void) free((char *) target);
}

/*
 * obj__update:
 *	Update the object list if necessary
 */
static
void
obj__update(proj)
OBJECT *proj;
{
	register OBJECT	*op, *prev, *next;
	long		last_change;
	void		read_objects();

	/*
	 * Hope that nothing has changed since we last checked
	 */
	if (prop_get((OFS_OBJECT) proj, "update", (void *) &last_change) < 0)
```

```
		return;
	if (last_change == proj->last_change)
		return;

	/*
	 * Something has changed since last time.  We have to actually
	 * reread the whole list of objects again
	 */
	proj->last_change = last_change;

	/*
	 * First we mark all objects for deletion since that's the
	 * only way we can detect whether an object disappeared.
	 * Then we actually read the project table to get all the
	 * objects.  Finally, we delete all those objects who are
	 * "still" marked for deletion.
	 */
	for (op = proj->objects; op != NULL; op = op->next)
		op->delete = TRUE;
	read_objects(proj);
	prev = NULL;
	for (op = proj->objects; op != NULL; op = next) {
		next = op->next;
		if (op->delete) {
			if (prev == NULL)
				proj->objects = next;
			else
				prev->next = next;
			obj__free(op);
		}
		else
			prev = op;
	}
}

/*
 * read_objects:
 *	Read the objects from the project table
 */
static
void
read_objects(proj)
OBJECT *proj;
{
	register int		i, n;
	register OBJECT	*op;
	DM_ATTR		name, type, class;
	char			oname[BUFLEN];
	short			otype, oclass;

	/*
	 * First we open up the attributes so we can get the info
	 */
	name = attr_open(proj->obj_list, "name");
	type = attr_open(proj->obj_list, "type");
	class = attr_open(proj->obj_list, "class");
	if (name == NULL || type == NULL || class == NULL) {
		dm_errno = DM_ESYSERROR;
		return;
	}

	/*
	 * Finally, loop through all the rows of the table
	 * and check the objects for existence
	 */
	n = cell_count(name, DM_I_MAX);
	for (i = 0; i <= n; i++) {
		if (cell_read(name, i, (void *) oname) < 0)
			continue;
		for (op = proj->objects; op != NULL; op = op->next) {
			if (strcmp(op->name, oname) != 0)
				continue;
			op->delete = FALSE;
			break;
		}
		if (op != NULL)
			continue;
		if (cell_read(class, i, (void *) &oclass) < 0)
			continue;
		if (cell_read(type, i, (void *) &otype) < 0)
			continue;
		(void) obj__add(proj, proj->dmh, oname, oclass, otype, i);
	}

	/*
	 * Release all our opened stuff
	 */
	(void) attr_close(name);
	(void) attr_close(type);
	(void) attr_close(class);
}

/*
 * obj_count:
```

```
 *	Count up the number of objects in a project
 */
int
obj_count(proj, oname, oclass, otype)
OFS_OBJECT	proj;
char		*oname;
int		oclass, otype;
{
	register OBJECT	*op, *pp;
	register int	count;

	if (proj == NULL) {
		dm_errno = DM_EBADPROJ;
		return -1;
	}
	pp = (OBJECT *) proj;
	obj__update(pp);		/* Make sure info is current */
	count = 0;
	for (op = pp->objects; op != NULL; op = op->next) {
		if ((op->class != oclass && oclass != -1)
		||  (op->type != otype && otype != -1)
		||  (oname != NULL && strcmp(op->name, oname) != 0))
			continue;
		count++;
	}
	return count;
}

/*
 * obj_list:
 *	List all objects in a project
 */
int
obj_list(proj, oname, oclass, otype, desc, n)
OFS_OBJECT	proj;
char		*oname;
int		oclass, otype;
OFS_OBJ_DESC	*desc;
int		n;
{
	register OBJECT	*op;
	register int	i;

	if (proj == NULL) {
		dm_errno = DM_EBADPROJ;
		return -1;
	}
	obj__update((OBJECT *) proj);
	i = 0;
	for (op = OBJ_OBJECTS(proj); op != NULL && i <= n; op = op->next) {
		if ((op->class != oclass && oclass != -1)
		||  (op->type != otype && otype != -1)
		||  (oname != NULL && strcmp(op->name, oname) != 0))
			continue;
		if (i >= n) {
			dm_errno = DM_ETOOSMALL;
			return -1;
		}
		desc[i].class = op->class;
		desc[i].type = op->type;
		desc[i].name = op->name;
		desc[i].ident = op->row;
		i++;
	}
	return i;
}

/*
 * get_obj:
 *	Given an object, make sure the associated table is open
 *	and increment the reference count
 */
static
int
get_obj(op)
OBJECT *op;
{
	DM_ATTR		ah;
	DM_TABLE	obj_th;

	/*
	 * If this object is not opened yet, fetch the table handle
	 */
	if (op->ref_count <= 0) {
		if ((ah = attr_open(op->parent->obj_list, "objects")) == NULL) {
			dm_errno = DM_ESYSERROR;
			return -1;
		}
		if (cell_read(ah, op->row, (void *) &obj_th) < 0) {
			(void) attr_close(ah);
			dm_errno = DM_ENOSUCH;
			return -1;
		}
		((TABLE *) obj_th)->can_cache = FALSE;
```

```
		(void) attr_close(ah);
		op->table = obj_th;
		op->ref_count = 0;
		if (op->class == OBJ_CL_PROJECT
		|| op->class == OBJ_CL_COMPOSITE) {
			if (prop_get((OFS_OBJECT) op, "obj_list",
			(void *) &op->obj_list) < 0) {
				dm_errno = DM_ESYSERROR;
				return -1;
			}
		}
		else
			op->obj_list = NULL;
	}

	/*
	 * Now increment the reference count and return the object handle
	 */
	op->ref_count++;
	return 0;
}

/*
 * obj_open:
 *	Find an object by its class, type and name
 */
OFS_OBJECT
obj_open(proj, oname, oclass, otype)
OFS_OBJECT	proj;
char		*oname;
int		oclass, otype;
{
	register OBJECT	*op, *save;

	if (proj == NULL
	|| (OBJ_CLASS(proj) != OBJ_CL_PROJECT
	&&  OBJ_CLASS(proj) != OBJ_CL_COMPOSITE)) {
		dm_errno = DM_EBADPROJ;
		return NULL;
	}

	/*
	 * Make sure the list of objects is up to date
	 */
	obj__update((OBJECT *) proj);

	/*
	 * First we find the object handle itself
	 */
	save = NULL;
	for (op = OBJ_OBJECTS(proj); op != NULL; op = op->next) {
		if ((op->class != oclass && oclass != -1)
		|| (op->type != otype && otype != -1)
		|| (oname != NULL && strcmp(op->name, oname) != 0))
			continue;
		if (save != NULL) {
			dm_errno = DM_ETOOMANY;
			return NULL;
		}
		save = op;
	}
	if (save == NULL) {
		dm_errno = DM_ENOSUCH;
		return NULL;
	}

	/*
	 * Make sure the object is properly accessible and return
	 */
	if (get_obj(save) < 0)
		return NULL;
	return (OFS_OBJECT) save;
}

/*
 * obj_open_by_ident:
 *	Find an object by its unique identifier
 */
OFS_OBJECT
obj_open_by_ident(proj, ident)
OFS_OBJECT	proj;
int		ident;
{
	register OBJECT	*op;

	if (proj == NULL
	|| (OBJ_CLASS(proj) != OBJ_CL_PROJECT
	&&  OBJ_CLASS(proj) != OBJ_CL_COMPOSITE)) {
		dm_errno = DM_EBADPROJ;
		return NULL;
	}

	/*
	 * Make sure the list of objects is up to date
```

```
	 */
	obj__update((OBJECT *) proj);

	/*
	 * First we find the object handle itself
	 */
	for (op = OBJ_OBJECTS(proj); op != NULL; op = op->next)
		if (op->row == ident)
			break;
	if (op == NULL) {
		dm_errno = DM_ENOSUCH;
		return NULL;
	}

	/*
	 * Make sure the object is properly accessible and return
	 */
	if (get_obj(op) < 0)
		return NULL;
	return (OFS_OBJECT) op;
}

/*
 * obj_create:
 *	Create an object of the given type in the given object.  This
 *	routine will look in the prototype project for a possibly registered
 *	object and copy it.  Or it will create and return a raw object.
 */
OFS_OBJECT
obj_create(proj, oname, oclass, otype)
OFS_OBJECT	proj;
char		*oname;
short		oclass, otype;
{
	OBJECT	*op;
	char	tname[BUFLEN];

	if (proj == NULL
	|| (OBJ_CLASS(proj) != OBJ_CL_PROJECT
	&& OBJ_CLASS(proj) != OBJ_CL_COMPOSITE)) {
		dm_errno = DM_EBADPROJ;
		return NULL;
	}

	dm__sanitize(oname, tname);
	op = obj__make(OBJ_DMH(proj), tname, oname, oclass, otype);
	if (op == NULL)
		return NULL;
	if (op->class == OBJ_CL_PROJECT || op->class == OBJ_CL_COMPOSITE) {
		if (prop_get((OFS_OBJECT) op, "obj_list",
		(void *) &op->obj_list) < 0) {
			obj__free(op);
			return NULL;
		}
	}
	if (obj__enter((OBJECT *) proj, op) < 0) {
		obj__free(op);
		return NULL;
	}
	return (OFS_OBJECT) op;
}

/*
 * obj_rename:
 *	Rename an object
 */
int
obj_rename(oh, newname)
OFS_OBJECT	oh;
char		*newname;
{
	DM_ATTR		ah;

	if (oh == NULL) {
		dm_errno = DM_EBADARG;
		return -1;
	}
	if ((ah = attr_open(OBJ_PARENT(oh)->obj_list, "name")) != NULL) {
		(void) cell_write(ah, OBJ_ROW(oh), (void *) newname);
		(void) attr_close(ah);
		return 0;
	}
	return -1;
}

/*
 * obj_close:
 *	Terminate access to this object
 */
int
obj_close(oh)
OFS_OBJECT	oh;
{
	OBJECT *op;
```

```
	int		n;

	op = (OBJECT *) oh;
	if (op == NULL || op->ref_count <= 0) {
		dm_errno = DM_EBADARG;
		return -1;
	}
	if (op->ref_count-- > 0)
		return 0;
	if (op->class == OBJ_CL_PROJECT || op->class == OBJ_CL_COMPOSITE
		if (op->obj_list != NULL)
			(void) tbl_close(op->obj_list);
	n = tbl_close(op->table);
	op->table = NULL;
	return n;
}

/*
 * obj_delete:
 *	Delete an object (no attempt is made to clean up lower
 *	level tables; higher order objects should have their own
 *	delete routines that cleans up before calling obj_delete)
 */
int
obj_delete(oh)
OFS_OBJECT	oh;
{
	register OBJECT	*parent;
	void		delete_table_tree();
	void		obj__remove();
	void		obj__fix_update();

	if (oh == NULL) {
		dm_errno = DM_EBADARG;
		return -1;
	}
	/*
	 * If there are other references to this object within this
	 * process, just fail immediately
	 */
	if (OBJ_REFCNT(oh) > 1) {
		dm_errno = DM_ECANTLOCK;
		return -1;
	}

	/*
	 * This process is okay.
	 */
	delete_table_tree(OBJ_TABLE(oh));

	/*
	 * Okay, so this object is gone.  We must go up to the parent,
	 * remove the object from the linked list, and update the
	 * "update" attribute of the parent
	 */
	parent = OBJ_PARENT(oh);
	if (parent == NULL) {
		obj__free((OBJECT *) oh);
		return 0;
	}
	obj__remove(parent, (OBJECT *) oh);
	return 0;
}

/*
 * delete_table_tree:
 *	Delete a tree of tables
 */
static
void
delete_table_tree(th)
DM_TABLE	th;
{
	register int	i, nattr;
	register int	j, ninst;
	DM_TABLE	subth;
	DM_ATTR		ah;
	DM_STYPE	stype;
	char		**aname, *anamebuf;

	/*
	 * Count the number of attributes.  If there are none, we just
	 * delete the table and return
	 */
	nattr = attr_count(th, DM_A_EXIST);
	if (nattr <= 0) {
		(void) tbl_delete(th);
		return;
	}

	/*
	 * Get the list of attributes
	 */
	aname = (char **) malloc((unsigned) nattr * sizeof (char *));
```

```
	anamebuf = (char *) malloc((unsigned) nattr * DM_ATTR_NAMELEN);
	if (aname == NULL || anamebuf == NULL) {
		dm_errno = DM_EOUTOFMEM;
		if (aname != NULL)
			(void) free((char *) aname);
		(void) tbl_delete(th);
		return;
	}
	for (i = 0; i < nattr; i++)
		aname[i] = anamebuf + i * DM_ATTR_NAMELEN;
	if (attr_list(th, DM_A_EXIST, aname, nattr) != nattr) {
		dm_errno = DM_ESYSERROR;
		(void) free((char *) aname);
		(void) free(anamebuf);
		(void) tbl_delete(th);
		return;
	}

	/*
	 * Loop through the attributes and recursively delete any
	 * tables contained in this table
	 */
	for (i = 0; i < nattr; i++) {
		ah = attr_open(th, aname[i]);
		if (ah == NULL)
			continue;
		stype = attr_stype(ah);
		if (stype.base_type != DM_S_TABLE) {
			(void) attr_close(ah);
			continue;
		}
		ninst = cell_count(ah, DM_I_MAX) + 1;
		for (j = 0; j < ninst; j++) {
			if (cell_read(ah, j, (void *) &subth) < 0)
				continue;
			delete_table_tree(subth);
		}
		(void) attr_close(ah);
	}

	/*
	 * Everything is clean below us, just clean up, delete
	 * this table and return
	 */
	(void) free((char *) aname);
	(void) free(anamebuf);
	(void) tbl_delete(th);
		return;
}

/*
 * obj_type:
 *		Return object type
 */
int
obj_type(oh)
OFS_OBJECT	oh;
{
		if (oh == NULL) {
			dm_errno = DM_EBADARG;
			return -1;
		}
		return OBJ_TYPE(oh);
}

/*
 * obj_class:
 *		Return object class
 */
int
obj_class(oh)
OFS_OBJECT	oh;
{
		if (oh == NULL) {
			dm_errno = DM_EBADARG;
			return -1;
		}
		return OBJ_CLASS(oh);
}

/*
 * obj_ident:
 *		Return object identifier
 */
int
obj_ident(oh)
OFS_OBJECT	oh;
{
		if (oh == NULL) {
			dm_errno = DM_EBADARG;
			return NULL;
		}
		return OBJ_IDENT(oh);
}
```

libofs/object.c

```
/*
 * obj_name:
 *	Return object name
 */
char *
obj_name(oh)
OFS_OBJECT	oh;
{
	if (oh == NULL) {
		dm_errno = DM_EBADARG;
		return NULL;
	}
	return OBJ_NAME(oh);
}

/*
 * obj_dmh:
 *	Return object data manager handle
 */
DM_HANDLE
obj_dmh(oh)
OFS_OBJECT	oh;
{
	if (oh == NULL) {
		dm_errno = DM_EBADARG;
		return NULL;
	}
	return OBJ_DMH(oh);
}

/*
 * obj_lock:
 *	Lock an object
 */
DM_KEY
obj_lock(oh, mode, key, wait)
OFS_OBJECT	oh;
DM_LMODE	mode;
DM_KEY		key;
int		wait;
{
	if (oh == NULL) {
		dm_errno = DM_EBADARG;
		return -1;
	}
	return tbl_lock(OBJ_TABLE(oh), mode, key, wait);
}

/*
 * obj_lock_change:
 *	Change the mode of a lock
 */
int
obj_lock_change(oh, mode, key, wait)
OFS_OBJECT	oh;
DM_LMODE	mode;
DM_KEY		key;
int		wait;
{
	if (oh == NULL) {
		dm_errno = DM_EBADARG;
		return -1;
	}
	return tbl_lock_change(OBJ_TABLE(oh), mode, key, wait);
}

/*
 * obj_unlock:
 *	Release the lock on an object
 */
int
obj_unlock(oh, key)
OFS_OBJECT	oh;
DM_KEY		key;
{
	if (oh == NULL) {
		dm_errno = DM_EBADARG;
		return -1;
	}
	return tbl_unlock(OBJ_TABLE(oh), key);
}

/*
 * obj_open_parent:
 *	Open parent of object
 */
OFS_OBJECT
obj_open_parent(oh)
OFS_OBJECT	oh;
{
	if (oh == NULL) {
		dm_errno = DM_EBADARG;
		return NULL;
```

```
	}

	If (OBJ_PARENT(oh) == NULL) {
		dm_ermo = DM_ENOSUCH;
		return NULL;
	}

	If (get_obj(OBJ_PARENT(oh)) < 0)
		return NULL;

	return (OFS_OBJECT) OBJ_PARENT(oh);
}
```

libofs/ofs.h

```
/*
 * $Header: ofs.h,v 1.2 88/10/21 16:22:52 conrad Locked $
 */
#ifndef OFS_INCLUDE
#define	OFS_INCLUDE

#include <dm.h>
#include <dm_class.h>

typedef	void	*OFS_OBJECT;
typedef	void	*OFS_PROP;

typedef struct {
	int	type, class;
	char	*name;
	int	ident;
}		OFS_OBJ_DESC;

typedef struct {
	int	type, class;
	char	*name;
	char	*description;
	char	*owner;
}		OFS_OBJ_TYPE;

typedef struct {
	int	class;
	char	*name;
	char	*description;
	char	*owner;
}		OFS_OBJ_CLASS;

#define	OFS_PROP_NAMELEN	DM_TBL_NAMELEN
#define	OFS_OBJ_NAMELEN	DM_ATTR_NAMELEN

/*
 * Things shared between DM and OFS
 */
#define	ofs_error	dm_error
#define	ofs_errno	dm_errno
#define	ofs_errlist	dm_errlist

/*
 * Access functions
 */
extern OFS_OBJECT	ofs_proj_init();
extern OFS_OBJECT	ofs_sys_init();

extern OFS_OBJECT	proj_open();
extern OFS_OBJECT	proj_create();
extern int	proj_delete();
extern int	proj_close();
extern char	*proj_id();

extern int	obj_count();
extern int	obj_list();
extern OFS_OBJECT	obj_open();
extern OFS_OBJECT	obj_open_by_ident();
extern OFS_OBJECT	obj_create();
extern int	obj_close();
extern int	obj_delete();
extern int	obj_type();
extern int	obj_class();
extern char	*obj_name();
extern DM_HANDLE	obj_dmh();
extern DM_KEY	obj_lock();
extern int	obj_lock_change();
extern int	obj_unlock();

extern int	prop_count();
extern int	prop_list();
extern DM_STYPE	prop_stype();
extern OFS_PROP	prop_open();
extern OFS_PROP	prop_create();
extern int	prop_delete();
extern int	prop_close();
extern int	prop_read();
extern int	prop_write();
extern int	prop_set();
extern int	prop_get();
extern DM_KEY	prop_lock();
extern int	prop_lock_change();
extern int	prop_unlock();

/*
 * Package variables
 */
extern OFS_OBJECT	proj__sys;

#endif
```

libofs/path.c

```c
/*
 * $Header: path.c,v 1.6 89/03/01 20:10:06 pett Exp $
 * $Log:	path.c,v $
 * Revision 1.6  89/03/01  20:10:06  pett
 * declared malloc as returning MALLOCTYPE
 *
 * Revision 1.5  88/10/17  11:53:18  conrad
 * Don't delay write's on TCP connections
 *
 * Revision 1.4  88/09/19  17:14:59  conrad
 * Use new dispatcher library interface
 *
 * Revision 1.3  88/09/07  15:01:37  conrad
 * General cleanup
 *
 * Revision 1.2  88/08/08  09:47:48  conrad
 * Standardize buffer sizes
 * Add support for STRING and TABLE types
 *
 * Revision 1.1  88/07/29  15:59:02  conrad
 * Initial revision
 *
 */
#include "dm_internal.h"
#include "handshake.h"
#include "response.h"
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

#define	MIN_VERS	1
#define	MAX_VERS	1

/*
 * dm_open_path:
 *	Open a path to the data manager
 */
DM_HANDLE
dm_open_path(cid, acct, host, path)
char	*cid, *acct, *host, *path;
{
	int		status;
	char		buf[BUFLEN];
	HANDLE		*hp;
	HANDLE		*get_handle();
	void		free_handle();

	if ((hp = get_handle(cid, acct, host, path)) == NULL)
		return NULL;

	(void) sprintf(buf, "OPEN DATASET \"%s\"\n", path);
	(void) write(fileno(hp->fp), buf, strlen(buf));
	status = mmwb_resp_ascii(hp->fp, buf, sizeof buf);
	if (dm__check_reply(status) < 0) {
		if (status == R_UNAVAIL)
			(void) strcpy(path, buf);
		free_handle(hp);
		return NULL;
	}
	return (DM_HANDLE) hp;
}

/*
 * free_handle:
 *	Release handle structure
 */
static
void
free_handle(hp)
HANDLE *hp;
{
	(void) dispatcher_server_close(hp->server);
	(void) fclose(hp->fp);
	(void) free(hp->name);
	(void) free((char *) hp);
}

/*
 * get_handle:
 *	Get a handle for the remote data manager
 */
static
HANDLE *
get_handle(cid, acct, host, path)
char	*cid;
char	*acct;
char	*host;
char	*path;
{
	int		fd;
	FILE		*fp;
	HANDLE		*hp;
	int		version, data_mode;
```

libofs/path.c

```c
	DISPATCHER_SERVER	server;
	extern MALLOCTYPE	malloc();
	int			machine();
#ifdef TCP_NODELAY
	int			on = 1;
#endif

	/*
	 * Get a file stream to the data manager
	 */
	server = dispatcher_call(host, acct, cid, DM_SERVICE);
	if (server == NULL) {
		dm_errno = DM_EDISPATCHER;
		return NULL;
	}
	fd = dispatcher_server_fd(server);
#ifdef TCP_NODELAY
	if (setsockopt(fd, IPPROTO_TCP, TCP_NODELAY, &on, sizeof on) < 0)
		perror("setsockopt");
#endif
	fp = fdopen(fd, "r+");
	if (fp == NULL) {
		dm_errno = DM_ESYSERROR;
		(void) close(fd);
		return NULL;
	}

	/*
	 * Perform basic handshaking
	 */
	if (mmwb_handshake(fp, acct, cid, MIN_VERS, MAX_VERS, &version)
	!= MMWB_HS_SUCCESS) {
		dm_errno = DM_EHANDSHAKE;
		(void) dispatcher_server_close(server);
		(void) fclose(fp);
		return NULL;
	}

	/*
	 * Send MACHINE command
	 */
	if (machine(fp, &data_mode) < 0) {
		dm_errno = DM_EPROTOCOL;
		(void) dispatcher_server_close(server);
		(void) fclose(fp);
		return NULL;
	}

	/*
	 * Handshake successful, return handle
	 */
	hp = (HANDLE *) malloc(sizeof (HANDLE));
	if (hp == NULL) {
		dm_errno = DM_EOUTOFMEM;
		(void) dispatcher_server_close(server);
		(void) fclose(fp);
		return NULL;
	}
	hp->name = (char *) malloc((unsigned) strlen(acct) + strlen(host)
		+ strlen(path) + 3);
	if (hp->name == NULL) {
		dm_errno = DM_EOUTOFMEM;
		(void) dispatcher_server_close(server);
		(void) fclose(fp);
		(void) free((char *) hp);
		return NULL;
	}
	(void) sprintf(hp->name, "%s@%s:%s", acct, host, path);
	hp->fp = fp;
	hp->version = version;
	hp->data_mode = data_mode;
	hp->atime = hp->mtime = 0;
	hp->server = server;
	return hp;
}

/*
 * machine:
 *	Send the machine command and set the data mode
 */
static
int
machine(fp, mode)
FILE	*fp;
int	*mode;
{
	char	buf[BUFLEN];

	(void) sprintf(buf, "MACHINE %s\n", MACHINE);
	(void) write(fileno(fp), buf, strlen(buf));
	switch (mmwb_resp_status(fp)) {
	  case R_OKAY:
		*mode = DM_ASCII;
		break;
```

libofs/path.c

```
		case R_ALTOKAY:
			*mode = DM_BINARY;
			break;
		default:
			return -1;
		}
		return 0;
}

/*
 * dm_close_path:
 *	Close a connection to a data manager
 */
int
dm_close_path(dmh)
DM_HANDLE	dmh;
{
	HANDLE *hp;
	int	status;
	void	free_handle();

	hp = (HANDLE *) dmh;
	(void) write(fileno(hp->fp), "QUIT\n", 5);
	status = mmwb_resp_status(hp->fp);
	free_handle(hp);
	if (status != R_GENOKAY) {
		dm_errno = DM_EPROTOCOL;
		return -1;
	}
	return 0;
}

/*
 * dm_create_path:
 *	Create a dataset
 */
DM_HANDLE
dm_create_path(cid, acct, host, path)
char	*cid;
char	*acct;
char	*host;
char	*path;
{
	int		status;
	char		buf[BUFLEN];
	HANDLE		*hp;
	HANDLE		*get_handle();
	void		free_handle();

	if ((hp = get_handle(cid, acct, host, path)) == NULL)
		return NULL;

	/*
	 * Create the dataset
	 */
	(void) sprintf(buf, "CREATE DATASET \"%s\"\n", path);
	(void) write(fileno(hp->fp), buf, strlen(buf));
	status = mmwb_resp_ascii(hp->fp, buf, sizeof buf);
	if (dm__check_reply(status) < 0) {
		free_handle(hp);
		return NULL;
	}

	/*
	 * Open the dataset
	 */
	(void) sprintf(buf, "OPEN DATASET \"%s\"\n", path);
	(void) write(fileno(hp->fp), buf, strlen(buf));
	status = mmwb_resp_ascii(hp->fp, buf, sizeof buf);
	if (dm__check_reply(status) < 0) {
		if (status == R_UNAVAIL)
			(void) strcpy(path, buf);
		free_handle(hp);
		return NULL;
	}
	return (DM_HANDLE) hp;
}

/*
 * dm_delete_path:
 *	Delete a dataset
 */
int
dm_delete_path(cid, acct, host, path)
char	*cid;
char	*acct;
char	*host;
char	*path;
{
	int		status;
	char		buf[BUFLEN];
	HANDLE		*hp;
	HANDLE		*get_handle();
	void		free_handle();
```

```
	if ((hp = get_handle(cid, acct, host, path)) == NULL)
		return -1;

	/*
	 * Open the dataset
	 */
	(void) sprintf(buf, "OPEN DATASET \"%s\"\n", path);
	(void) write(fileno(hp->fp), buf, strlen(buf));
	status = mmwb_resp_ascii(hp->fp, buf, sizeof buf);
	if (dm__check_reply(status) < 0) {
		if (status == R_UNAVAIL)
			(void) strcpy(path, buf);
		free_handle(hp);
		return -1;
	}

	/*
	 * Now delete it
	 */
	(void) sprintf(buf, "DELETE DATASET \"%s\"\n", path);
	(void) write(fileno(hp->fp), buf, strlen(buf));
	status = mmwb_resp_status(hp->fp);
	if (dm__check_reply(status) < 0) {
		free_handle(hp);
		return -1;
	}
	return dm_close_path((DM_HANDLE) hp);
}

/*
 * dm_times:
 *	Fill in the access and modification times of a dataset
 */
int
dm_times(dmh, atime, mtime)
DM_HANDLE	dmh;
time_t		*atime, *mtime;
{
	HANDLE *hp;
	int	get_path_desc();

	hp = (HANDLE *) dmh;
	if (get_path_desc(hp) < 0)
		return -1;
	*atime = hp->atime;
	*mtime = hp->mtime;
		return 0;
}

/*
 * get_path_desc:
 *	Get dataset description
 */
static
int
get_path_desc(hp)
HANDLE *hp;
{
	int	read_path_desc();

	(void) write(fileno(hp->fp), "DESCRIBE DATASET\n", 17);
	return dm__read_list(hp, read_path_desc, (void *) hp,
		(void *) NULL);
}

/*
 * read_path_desc:
 *	Read the actual dataset descriptions
 */
/* ARGSUSED */
static
int
read_path_desc(fp, arg1, arg2)
FILE	*fp;
void	*arg1, *arg2;
{
	register HANDLE	*hp;
	char		buf[BUFLEN];

	hp = (HANDLE *) arg1;
	while (fgets(buf, sizeof buf, fp) != NULL) {
		if (buf[0] == '.' && buf[1] == '\n')
			return 0;
		if (strncmp(buf, "time", 4) == 0)
			(void) sscanf(buf, "time %ld %ld", &hp->mtime,
				&hp->atime);
	}
	dm_errno = DM_EPROTOCOL;
	return -1;
}

/*
 * dm_path_id:
```

libofs/path.c

```
 *          Return the handle identifier for later reconnection
 */
char *
dm_path_id(dmh)
DM_HANDLE        dmh;
{
            return (((HANDLE *) dmh)->name);
}

/*
 * dm_path_join:
 *          Join a data manager via a path id
 */
DM_HANDLE
dm_path_join(cid, path_id)
char        *cid;
char        *path_id;
{
            char        host[BUFLEN], acct[BUFLEN], path[BUFLEN];

            dm__parse_path_id(path_id, acct, host, path);
            if (acct[0] == '\0' || host[0] == '\0' || path[0] == '\0') {
                        dm_errno = DM_EBADPATHID;
                        return NULL;
            }
            return dm_open_path(cid, acct, host, path);
}
```

libofs/project.c

```
#include "ofs.h"
#include "dm_internal.h"
#include <ctype.h>
#include <netdb.h>
#include <pwd.h>

/*
 * A project object has these additional properties:
 *	host: which host dataset lives on
 *	path: pathname on that host to use
 */

OFS_OBJECT	proj__sys;
static char	*proj_host, *proj_acct;

/*
 * proj_raw_open:
 *	Open a project by host and pathname
 */
OFS_OBJECT
proj_raw_open(acct, host, pathname)
char	*acct;
char	*host;
char	*pathname;
{
	DM_HANDLE	dmh;
	char		cid[BUFLEN];
	int		make_cid();

	/*
	 * Try to open the path
	 */
	if (make_cid(cid, host, pathname) < 0)
		return NULL;
	if ((dmh = dm_open_path(cid, acct, host, pathname)) == NULL)
		return NULL;

	/*
	 * We have all the information we need, just build the object
	 */
	return (OFS_OBJECT) obj__get(dmh, PROJECT_TABLE_NAME);
}

/*
 * proj_open:
 *	Open a sub-project
 */
OFS_OBJECT
proj_open(oh)
OFS_OBJECT	oh;		/* Handle of target project */
{
	char	path[BUFLEN], host[BUFLEN], acct[BUFLEN];
	int	set_acct();

	if (OBJ_TYPE(oh) != OBJ_T_PROJECT) {
		dm_errno = DM_EBADPROJ;
		return NULL;
	}
	if (prop_get(oh, "path", (void *) path) < 0)
		return NULL;
	if (prop_get(oh, "host", (void *) host) < 0)
		return NULL;
	if (set_acct(host, acct) < 0)
		return NULL;
	return proj_raw_open(acct, host, path);
}

/*
 * make_cid:
 *	Create a cid from a pathname and a host
 */
static
int
make_cid(cid, host, path)
char	*cid;
char	*host, *path;
{
	register char	*cp;
	struct hostent	*hp;

	if ((hp = gethostbyname(host)) == NULL) {
		dm_errno = DM_ENOSUCH;
		return -1;
	}
	(void) sprintf(cid, "%s@%s", path, hp->h_name);
	for (cp = cid; *cp != '\0'; cp++)
		if (isupper(*cp))
			*cp = tolower(*cp);
	return 0;
}

/*
 * proj_raw_create:
 *	Create a project by host and pathname
```

```
 */
OFS_OBJECT
proj_raw_create(acct, host, pathname, oname, phost, ppath)
char	*acct;
char	*host;
char	*pathname;
char	*oname;
char	*phost, *ppath;
{
	DM_HANDLE	dmh;
	char		cid[BUFLEN];
	int		make_cid();
	OFS_OBJECT	make_proj_obj();

	if (make_cid(cid, host, pathname) < 0)
		return NULL;
	if ((dmh = dm_create_path(cid, acct, host, pathname)) == NULL)
		return NULL;
	return make_proj_obj(dmh, PROJECT_TABLE_NAME, oname, host, pathn
		phost, ppath);
}

/*
 * make_proj_obj:
 *	Create a project object by the given name into a dataset
 */
static
OFS_OBJECT
make_proj_obj(dmh, tname, oname, host, path, phost, ppath)
DM_HANDLE	dmh;
char		*tname, *oname;
char		*host, *path;
char		*phost, *ppath;
{
	OFS_OBJECT	oh;

	oh = (OFS_OBJECT) obj__make(dmh, tname, oname, OBJ_CL_PROJECT,
		OBJ_T_PROJECT);
	if (oh == NULL)
		return NULL;
	if (prop_set(oh, "host", (void *) host) < 0)
		return NULL;
	if (prop_set(oh, "path", (void *) path) < 0)
		return NULL;
	if (phost != NULL && prop_set(oh, "parent_host", (void *) phost) < 0)
		return NULL;
	if (ppath != NULL && prop_set(oh, "parent_path", (void *) ppath) < 0)
		return NULL;
	return oh;
}

/*
 * proj_create:
 *	Create a sub-project
 */
OFS_OBJECT
proj_create(oh, oname, hname, pname)
OFS_OBJECT	oh;
char		*oname;
char		*hname, *pname;		/* New host/path names */
{
	OFS_OBJECT	proj;
	char		cur_host[BUFLEN], cur_path[BUFLEN];
	char		path[BUFLEN], acct[BUFLEN];
	char		buf[BUFLEN];
	int		set_acct();

	if (OBJ_TYPE(oh) != OBJ_T_PROJECT) {
		dm_errno = DM_EBADPROJ;
		return NULL;
	}

	if (prop_get(oh, "path", (void *) cur_path) < 0)
		return NULL;
	if (prop_get(oh, "host", (void *) cur_host) < 0)
		return NULL;

	if (set_acct(hname, acct) < 0)
		return NULL;
	if (hname == NULL)
		hname = cur_host;
	if (pname == NULL) {
		dm__sanitize(oname, buf);
		(void) sprintf(path, "%s/%s.project", cur_path, buf);
		pname = path;
	}

	proj = obj_create(oh, oname, OBJ_CL_PROJECT, OBJ_T_PROJECT);
	if (proj == NULL)
		return NULL;
	if (prop_set(proj, "host", (void *) hname) < 0)
		return NULL;
	if (prop_set(proj, "path", (void *) pname) < 0)
		return NULL;
```

```
	return proj_raw_create(acct, hname, pname, oname, cur_host, cur_path);
}

/*
 * proj_raw_delete:
 *	Delete a project by host and pathname
 */
int
proj_raw_delete(acct, host, pathname)
char	*acct;
char	*host;
char	*pathname;
{
	char			cid[BUFLEN];
	int			make_cid();

	if (make_cid(cid, host, pathname) < 0)
		return -1;
	return dm_delete_path(cid, acct, host, pathname);
}

/*
 * proj_delete:
 *	Delete a project when given a handle
 */
int
proj_delete(oh)
OFS_OBJECT	oh;
{
	char	path[BUFLEN], host[BUFLEN], acct[BUFLEN];
	void	delete_sub_projects();

	if (OBJ_TYPE(oh) != OBJ_T_PROJECT) {
		dm_errno = DM_EBADPROJ;
		return NULL;
	}

	/*
	 * We *only* delete sub-projects.  The biggest problem with
	 * deleting the project that the user is currently in, is that
	 * the user is then in limbo (no associated object).
	 */
	if (prop_get(oh, "parent_host", (void *) host) >= 0) {
		dm_errno = DM_ENOPERM;
		return NULL;
	}

	/*
	 * Okay.  We are deleting a sub-project.  We have to open
	 * it up and delete all *its* sub-projects.  Then we kill it.
	 */
	if (prop_get(oh, "path", (void *) path) < 0)
		return NULL;
	if (prop_get(oh, "host", (void *) host) < 0)
		return NULL;
	if (set_acct(host, acct) < 0)
		return NULL;

	/*
	 * Have all information needed from the object, kill the reference
	 * in the parent.  Then kill the sub-project and its descendents.
	 */
	(void) obj_delete(oh);
	delete_sub_projects(acct, host, path);
	return proj_raw_delete(acct, host, path);
}

/*
 * delete_sub_projects:
 *	Delete sub-projects from a project
 */
static
void
delete_sub_projects(acct, host, path)
char	*acct;
char	*host;
char	*path;
{
	register int		nproj, i;
	OFS_OBJECT		proj, sub;
	OFS_OBJ_DESC		*sub_list;
	extern MALLOCTYPE	malloc();

	proj = proj_raw_open(acct, host, path);
	if (proj == NULL)
		return;
	nproj = obj_count(proj, (char *) NULL, OBJ_CL_PROJECT, OBJ_T_PROJECT);
	if (nproj <= 0)
		return;
	sub_list = (OFS_OBJ_DESC *)
		malloc((unsigned) nproj * sizeof (OFS_OBJ_DESC));
	if (sub_list == NULL)
		return;
	(void) obj_list(proj, (char *) NULL, OBJ_CL_PROJECT, OBJ_T_PROJECT,
```

```
		sub_list, nproj);

	for (i = 0; i < nproj; i++) {
		sub = obj_open_by_ident(proj, sub_list[i].ident);
		if (sub != NULL)
			(void) proj_delete(sub);
	}

	(void) proj_close(proj);
	(void) free((char *) sub_list);
}

/*
 * proj_close:
 *	Close a project
 */
int
proj_close(oh)
OFS_OBJECT	oh;
{
	if (OBJ_TYPE(oh) != OBJ_T_PROJECT) {
		dm_errno = DM_EBADARG;
		return -1;
	}
	if (dm_close_path(OBJ_DMH(oh)) < 0)
		return -1;
	obj__free((OBJECT *) oh);		/* Can't release normally
						 * since dmh is gone */
	return 0;
}

/*
 * ofs_sys_init:
 *	Initialize data management system
 */
OFS_OBJECT
ofs_sys_init(id)
char	*id;
{
	OFS_OBJECT	oh;
	char		host[BUFLEN], path[BUFLEN], acct[BUFLEN];
	struct passwd	*pp;
	extern char	*getenv();
	extern UIDTYPE	geteuid();

	/*
	 * First we figure out where the MMWB project is
	 */
	if (id != NULL)
		dm__parse_path_id(id, acct, host, path);
	else
		dm__parse_path_id(getenv(MMWB_ENV), acct, host, path);
	if (acct[0] =='\0') {
		if ((pp = getpwuid(geteuid())) == NULL)
			(void) strcpy(acct, "nobody");
		else
			(void) strcpy(acct, pp->pw_name);
	}
	if (host[0] == '\0')
		(void) gethostname(host, sizeof host);
	if (path[0] == '\0')
		(void) strcpy(path, PROTOTYPE_PROJECT);

	/*
	 * Now we open up the prototype project on the target system
	 */
	oh = proj_raw_open(acct, host, path);
	if (oh != NULL) {
		proj_host = dm__copy_string(host);
		proj_acct = dm__copy_string(acct);
	}
	proj__sys = oh;
	return oh;
}

/*
 * ofs_proj_init:
 *	Initialize a user project
 */
OFS_OBJECT
ofs_proj_init(id)
char	*id;
{
	char		acct[BUFLEN], host[BUFLEN], path[BUFLEN];
	OFS_OBJECT	oh, proj;
	DM_HANDLE	dmh;
	char		cid[BUFLEN];
	int		make_cid();
	int		set_acct();

	/*
	 * Make sure the system prototype project is open
	 */
	if (proj__sys == NULL)
```

```
		if (ofs_sys_init((char *) NULL) == NULL)
			return NULL;

	/*
	 * If id is NULL, then we find and return the home project
	 */
	if (id == NULL) {
		if (set_acct((char *) NULL, acct) < 0)
			return NULL;
		oh = obj_open(proj__sys, acct, OBJ_CL_PROJECT, OBJ_T_PRO
		if (oh == NULL)
			return NULL;
		if ((proj = proj_open(oh)) == NULL) {
			(void) obj_close(oh);
			return NULL;
		}
		return proj;
	}

	/*
	 * We decode the id and get to a data manager for a project
	 */
	dm__parse_path_id(id, acct, host, path);
	if (acct[0] == '\0' || host[0] == '\0' || path[0] == '\0') {
		dm_errno = DM_EBADPATHID;
		return NULL;
	}
	if (make_cid(cid, host, path) < 0)
		return NULL;
	if ((dmh = dm_open_path(cid, acct, host, path)) == NULL)
		return NULL;

	/*
	 * We have all the information we need, just build the object
	 */
	return (OFS_OBJECT) obj__get(dmh, PROJECT_TABLE_NAME);
}

/*
 * set_acct:
 *	Set the account name
 */
static
int
set_acct(host, acct)
char	*host;
char	*acct;
{
	if (host == NULL || strcmp(host, proj_host) == 0)
		(void) strcpy(acct, proj_acct);
	else {
		/* TODO: Figure out account by host */
		(void) strcpy(acct, proj_acct);
	}
	return 0;
}

/*
 * proj_id:
 *	Return a project id that may be used to reconnect via "ofs_proj_init"
 */
char *
proj_id(proj)
OFS_OBJECT	proj;
{
	return dm_path_id(OBJ_DMH(proj));
}
```

libofs/property.c

```
#include "ofs.h"
#include "dm_internal.h"

/*
 * prop_count:
 *         Count number of properties
 */
int
prop_count(oh)
OFS_OBJECT      oh;
{
        return attr_count(OBJ_TABLE(oh), DM_A_EXIST);
}

/*
 * prop_list:
 *         List all properties
 */
int
prop_list(oh, names, n)
OFS_OBJECT      oh;
char            **names;
int             n;
{
        return attr_list(OBJ_TABLE(oh), DM_A_EXIST, names, n);
}

/*
 * prop_stype:
 *         Return the storage type of the given property
 */
DM_STYPE
prop_stype(ph)
OFS_PROP        ph;
{
        return attr_stype((DM_ATTR) ph);
}

/*
 * prop_open:
 *         Open a property
 */
OFS_PROP
prop_open(oh, prop)
OFS_OBJECT      oh;
char            *prop;
{
        return (OFS_PROP) attr_open(OBJ_TABLE(oh), prop);
}

/*
 * prop_create:
 *         Create a property
 */
OFS_PROP
prop_create(oh, prop, stype)
OFS_OBJECT      oh;
char            *prop;
DM_STYPE        stype;
{
        DM_ATTR         ah;
        DM_TABLE        th;

        ah = attr_create(OBJ_TABLE(oh), prop, stype);
        if (ah == NULL)
                return NULL;

        /*
         * Create an empty table if the storage type is table
         */
        if (stype.base_type == DM_S_TABLE) {
                th = tbl_create(ah, 0, DM_CL_NONE);
                if (th != NULL)
                        (void) tbl_close(th);
        }

        return (OFS_PROP) ah;
}

/*
 * prop_delete:
 *         Delete a property
 */
int
prop_delete(ph)
OFS_PROP        ph;
{
        return attr_delete((DM_ATTR) ph);
}

/*
 * prop_close:
 *         Close a property
 */
```

libofs/property.c

```
int
prop_close(ph)
OFS_PROP	ph;
{
	return attr_close((DM_ATTR) ph);
}

/*
 * prop_read:
 *	Read property value
 */
int
prop_read(ph, data)
OFS_PROP	ph;
void	*data;
{
	return cell_read((OFS_PROP) ph, 0, data);
}

/*
 * prop_write:
 *	Write property value
 */
int
prop_write(ph, data)
OFS_PROP	ph;
void	*data;
{
	return cell_write((OFS_PROP) ph, 0, data);
}

/*
 * prop_lock:
 *	Lock a property
 */
DM_KEY
prop_lock(ph, mode, key, wait)
OFS_PROP	ph;
int	mode;
DM_KEY key;
int	wait;
{
	return attr_lock((DM_ATTR) ph, mode, key, wait);
}

/*
 * prop_lock_change:
 *	Change the lock mode on a property
 */
DM_KEY
prop_lock_change(ph, mode, key, wait)
OFS_PROP	ph;
int		mode;
DM_KEY		key;
int		wait;
{
	return attr_lock_change((DM_ATTR) ph, mode, key, wait);
}

/*
 * prop_unlock:
 *	Unlock a property
 */
int
prop_unlock(ph, key)
OFS_PROP	ph;
DM_KEY		key;
{
	return attr_unlock((DM_ATTR) ph, key);
}

/*
 * prop_get:
 *	Get property value
 */
int
prop_get(oh, prop, data)
OFS_OBJECT	oh;
char		*prop;
void		*data;
{
	DM_ATTRah;
	int	val;

	if ((ah = attr_open(OBJ_TABLE(oh), prop)) == NULL) {
		dm_errno = DM_ENOSUCH;
		return -1;
	}
	val = cell_read(ah, 0, data);
	(void) attr_close(ah);
	return val;
}

/*
```

libofs/property.c

```
 * prop_set:
 *          Set property value
 */
int
prop_set(oh, prop, data)
OFS_OBJECT      oh;
char            *prop;
void            *data;
{
        DM_ATTR ah;
        int     val;

        if ((ah = attr_open(OBJ_TABLE(oh), prop)) == NULL) {
                dm_errno = DM_ENOSUCH;
                return -1;
        }
        val = cell_write(ah, 0, data);
        (void) attr_close(ah);
        return val;
}
```

libofs/sync.c

```
/*
 * $Header: sync.c,v 1.1 88/07/29 15:59:03 conrad Exp $
 * $Log:        sync.c,v $
 * Revision 1.1  88/07/29  15:59:03  conrad
 * Initial revision
 *
 */
#include "dm_internal.h"

/*
 * dm_sync:
 *          Flush local buffers out to data manager
 *          Currently a no-op
 */
/* ARGSUSED */
void
dm_sync(dmh)
DM_HANDLE       dmh;
{
        return;
}
```

libofs/table.c

```
/*
 * $Header: table.c,v 1.6 89/03/01 20:29:56 pett Exp $
 * $Log:        table.c,v $
 * Revision 1.6  89/03/01  20:29:56  pett
 * changed various declarations to take advantage of machine-independence
 * offered by machdep.h
 *
 * Revision 1.5  88/10/04  15:33:44  conrad
 * Add local data cache
 *
 * Revision 1.4  88/09/20  22:38:43  conrad
 * When creating a raw table, try with several names before giving up
 *
 * Revision 1.3  88/09/07  14:59:31  conrad
 * Add new function tbl_name
 * Lock and unlock table while listing
 *
 * Revision 1.2  88/08/08  09:47:50  conrad
 * Standardize buffer sizes
 * Add support for STRING and TABLE types
 *
 * Revision 1.1  88/07/29  15:59:04  conrad
 * Initial revision
 *
 */
#include "dm_internal.h"
#include "response.h"
#include <ctype.h>

/*
 * tbl_count:
 *      Count the number of tables
 */
int
tbl_count(dmh)
DM_HANDLE       dmh;
{
        register HANDLE *hp;
        int             n;

        hp = (HANDLE *) dmh;
        (void) write(fileno(hp->fp), "COUNT TABLE\n", 12);
        if (dm__resp_number(hp->fp, &n) < 0)
                return -1;
        return n;
}

/*
 * tbl_count_by_class:
 *      Count the number of tables by class
 */
int
tbl_count_by_class(dmh, class)
DM_HANDLE       dmh;
int             class;
{
        register HANDLE *hp;
        char            buf[BUFLEN];
        int             n;

        hp = (HANDLE *) dmh;
        (void) sprintf(buf, "COUNT TABLE CLASS %d\n", class);
        (void) write(fileno(hp->fp), buf, strlen(buf));
        if (dm__resp_number(hp->fp, &n) < 0)
                return -1;
        return n;
}

/*
 * tbl_list:
 *      Get a list of tables
 */
int
tbl_list(dmh, names, n)
DM_HANDLE       dmh;
char            *names[];
int             n;
{
        HANDLE *hp;
        int     read_names();

        hp = (HANDLE *) dmh;
        (void) write(fileno(hp->fp), "LIST TABLE\n", 11);
        return dm__read_list(hp, read_names, (void *) names, (void *) &n);
}

/*
 * tbl_list_by_class:
 *      Get a list of tables of a certain class
 */
int
tbl_list_by_class(dmh, class, names, n)
DM_HANDLE       dmh;
int             class;
```

libofs/table.c

```
char			*names[];
int			n;
{
	HANDLE	*hp;
	char	buf[BUFLEN];
	int	read_names();

	hp = (HANDLE *) dmh;
	(void) sprintf(buf, "LIST TABLE CLASS %d\n", class);
	(void) write(fileno(hp->fp), buf, strlen(buf));
	return dm__read_list(hp, read_names, (void *) names, (void *) &n);
}

/*
 * read_names:
 *	Read the actual tables names
 */
static
int
read_names(fp, arg1, arg2)
FILE	*fp;
void	*arg1, *arg2;
{
	register int		i;
	register char		*cp;
	char			**names;
	int			n;
	char			buf[BUFLEN];

	names = (char **) arg1;
	n = *(int *) arg2;
	i = 0;
	while (fgets(buf, sizeof buf, fp) != NULL) {
		if (buf[0] == '.' && buf[1] == '\n') {
			if (i > n) {
				dm_errno = DM_ETOOSMALL;
				return -1;
			}
			return i;
		}
		if (i < n) {
			if ((cp = strchr(buf, '\n')) != NULL)
				*cp = '\0';
			(void) strcpy(names[i], buf);
		}
		i++;
	}
		dm_errno = DM_EPROTOCOL;
		return -1;
}

/*
 * tbl_open:
 *	Open a table and return table handle
 */
DM_TABLE
tbl_open(dmh, name)
DM_HANDLE	dmh;
char		*name;
{
	register HANDLE		*hp;
	register TABLE		*tp;
	char			buf[BUFLEN];
	extern MALLOCTYPE	malloc();

	hp = (HANDLE *) dmh;
	(void) sprintf(buf, "OPEN TABLE \"%s\"\n", name);
	(void) write(fileno(hp->fp), buf, strlen(buf));
	if (dm__check_reply(mmwb_resp_status(hp->fp)) < 0)
		return NULL;

	tp = (TABLE *) malloc(sizeof (TABLE));
	if (tp == NULL) {
		dm_errno = DM_EOUTOFMEM;
		return NULL;
	}
	tp->name = (char *) malloc((unsigned) strlen(name) + 1);
	if (tp->name == NULL) {
		(void) free((char *) tp);
		dm_errno = DM_EOUTOFMEM;
		return NULL;
	}
	(void) strcpy(tp->name, name);
	tp->hp = hp;
	tp->attr = NULL;
	tp->lock = NULL;
	tp->key = -1;
	tp->status = 0;
	tp->can_cache = TRUE;

	return (DM_TABLE) tp;
}

/*
```

libofs/table.c

```
 * tbl_close:
 *         Close a table
 */
int
tbl_close(th)
DM_TABLE        th;
{
        register TABLE  *tp;
        int             val;
        char            buf[BUFLEN];
        void            free_table();

        tp = (TABLE *) th;
        (void) sprintf(buf, "CLOSE TABLE \"%s\"\n", tp->name);
        (void) write(fileno(tp->hp->fp), buf, strlen(buf));
        val = dm__check_reply(mmwb_resp_status(tp->hp->fp));
        free_table(tp);
        return val;
}

/*
 * free_table:
 *         Release the table data structure
 */
static
void
free_table(tp)
TABLE   *tp;
{
        register LOCK   *lp, *nextlp;
        register ATTR   *ap, *nextap;

        (void) free(tp->name);
        for (lp = tp->lock; lp != NULL; lp = nextlp) {
                nextlp = lp->next;
                (void) free((char *) lp);
        }
        for (ap = tp->attr; ap != NULL; ap = nextap) {
                nextap = ap->next;
                attr__free(ap);
        }
        (void) free((char *) tp);
}

/*
 * tbl_raw_create:
 *         Create and open a table "by name" and return table handle
 */
DM_TABLE
tbl_raw_create(dmh, name, class)
DM_HANDLE       dmh;
char            *name;
int             class;
{
        register HANDLE *hp;
        char            buf[BUFLEN];
        int             alter_name();

        hp = (HANDLE *) dmh;
        for (;;) {
                (void) sprintf(buf, "CREATE TABLE \"%s\" %d\n", name, class);
                (void) write(fileno(hp->fp), buf, strlen(buf));
                if (dm__check_reply(mmwb_resp_status(hp->fp)) == 0)
                        break;
                if (dm_errno != DM_EEXISTS)
                        return NULL;
                if (alter_name(name) < 0) {
                        dm_errno = DM_ESYSERROR;
                        return NULL;
                }
        }
        return tbl_open(dmh, name);
}

/*
 * alter_name:
 *         Alter the given table name slightly so it might be unique
 */
static
int
alter_name(name)
char    *name;
{
        register char   *cp;

        for (cp = name + strlen(name) - 1; cp >= name; cp--)
                if (*cp == '.')
                        continue;
                else if (isalnum(*cp + 1)) {
                        *cp = *cp + 1;
                        return 0;
                }
        return -1;
}
```

libofs/table.c

```
/*
 * tbl_create:
 *	Create and open a table at a cell
 */
DM_TABLE
tbl_create(ah, row, class)
DM_ATTRah;
int	row;
int	class;
{
	register ATTR	*ap;
	DM_TABLE	th;
	char		buf[BUFLEN];

	ap = (ATTR *) ah;
	tbl__make_name(buf, ap->tp->name, ap->name, row);
	th = tbl_raw_create((DM_HANDLE) ap->tp->hp, buf, class);
	if (th == NULL)
		return NULL;
	if (cell_write(ah, row, (void *) &th) < 0) {
		(void) tbl_delete(th);
		return NULL;
	}
	return th;
}

/*
 * tbl__make_name:
 *	Make a table name from a table name, an attribute name and a row
 */
void
tbl__make_name(buf, tname, aname, row)
char	*buf;
char	*tname, *aname;
int	row;
{
	while (*tname != '\0') {
		*buf++ = *tname++;
		if (*tname == '.') {
			*buf++ = '.';
			break;
		}
	}
	*buf++ = '.';
	while (*aname != '\0')
		*buf++ = *aname++;
		*buf++ = '_';
		(void) sprintf(buf, "%d", row);
}

/*
 * tbl_delete:
 *	Delete a table and close it (assumes that the appropriate locks
 *	are already in place, for now)
 */
int
tbl_delete(th)
DM_TABLE	th;
{
	register TABLE	*tp;
	char		buf[BUFLEN];
	void		free_table();

	tp = (TABLE *) th;
	if (tbl_lock(th, DM_WRITELOCK, DEFAULT_KEY, TRUE) < 0)
		return -1;
	(void) sprintf(buf, "DELETE TABLE \"%s\"\n", tp->name);
	(void) write(fileno(tp->hp->fp), buf, strlen(buf));
	if (dm__check_reply(mmwb_resp_status(tp->hp->fp)) < 0)
		return -1;
	free_table(tp);
	return 0;
}

/*
 * tbl_class:
 *	Get the class of a table
 */
int
tbl_class(th)
DM_TABLE	th;
{
	register TABLE	*tp;
	int		get_tbl_desc();

	tp = (TABLE *) th;
	if ((tp->status & TBL_ST_GOTDESC) == 0)
		if (get_tbl_desc(tp) < 0)
			return -1;
	return tp->class;
}

/*
```

```
 * get_tbl_desc:
 *         Get table description
 */
static
int
get_tbl_desc(tp)
TABLE   *tp;
{
        char    buf[BUFLEN];
        int     read_tbl_desc();

        if (tbl_lock((DM_TABLE) tp, DM_READLOCK, DEFAULT_KEY, TRUE) < 0)
                return -1;
        (void) sprintf(buf, "DESCRIBE TABLE \"%s\"\n", tp->name);
        (void) write(fileno(tp->hp->fp), buf, strlen(buf));
        (void) dm__read_list(tp->hp, read_tbl_desc, (void *) tp, (void *) NULL);
        (void) tbl_unlock((DM_TABLE) tp, DEFAULT_KEY);
        tp->status |= TBL_ST_GOTDESC;
        return 0;
}

/*
 * read_tbl_desc:
 *         Read the actual tables descriptions
 */
/* ARGSUSED */
static
int
read_tbl_desc(fp, arg1, arg2)
FILE    *fp;
void    *arg1, *arg2;
{
        register TABLE  *tp;
        char            buf[BUFLEN];
        char            owner[BUFLEN];

        tp = (TABLE *) arg1;
        while (fgets(buf, sizeof buf, fp) != NULL) {
                if (buf[0] == '.' && buf[1] == '\n')
                        return 0;
                if (strncmp(buf, "instances", 9) == 0)
                        (void) sscanf(buf, "instances %d", &tp->max_inst);
                else if (strncmp(buf, "time", 4) == 0)
                        (void) sscanf(buf, "time %ld %ld", &tp->mtime,
                                &tp->atime);
                else if (strncmp(buf, "attributes", 10) == 0)
                        (void) sscanf(buf, "attributes %d %d %d",
```

```
                                &tp->nattr_exist, &tp->nattr_filled,
                                &tp->nattr_full);

                if (tp->status & TBL_ST_GOTDESC)
                        continue;
                /* These descriptions only need to be read once */
                if (strncmp(buf, "owner", 5) == 0) {
                        if (tp->owner == NULL) {
                                (void) sscanf(buf, "owner %s", owner);
                                tp->owner = dm__copy_string(owner);
                        }
                }
                else if (strncmp(buf, "class", 5) == 0)
                        (void) sscanf(buf, "class %d", &tp->class);
        }
        dm_errno = DM_EPROTOCOL;
        return -1;
}

/*
 * tbl_name:
 *         Return name of table
 */
char *
tbl_name(th)
DM_TABLE        th;
{
        return ((TABLE *) th)->name;
}

/*
 * tbl_lock:
 *         Lock a table
 */
int
tbl_lock(th, mode, key, wait)
DM_TABLE        th;
int             mode;
int             key;
int             wait;
{
        register TABLE  *tp;
        int             state;
        char            *command;
        int             get_tbl_lock();

        tp = (TABLE *) th;
```

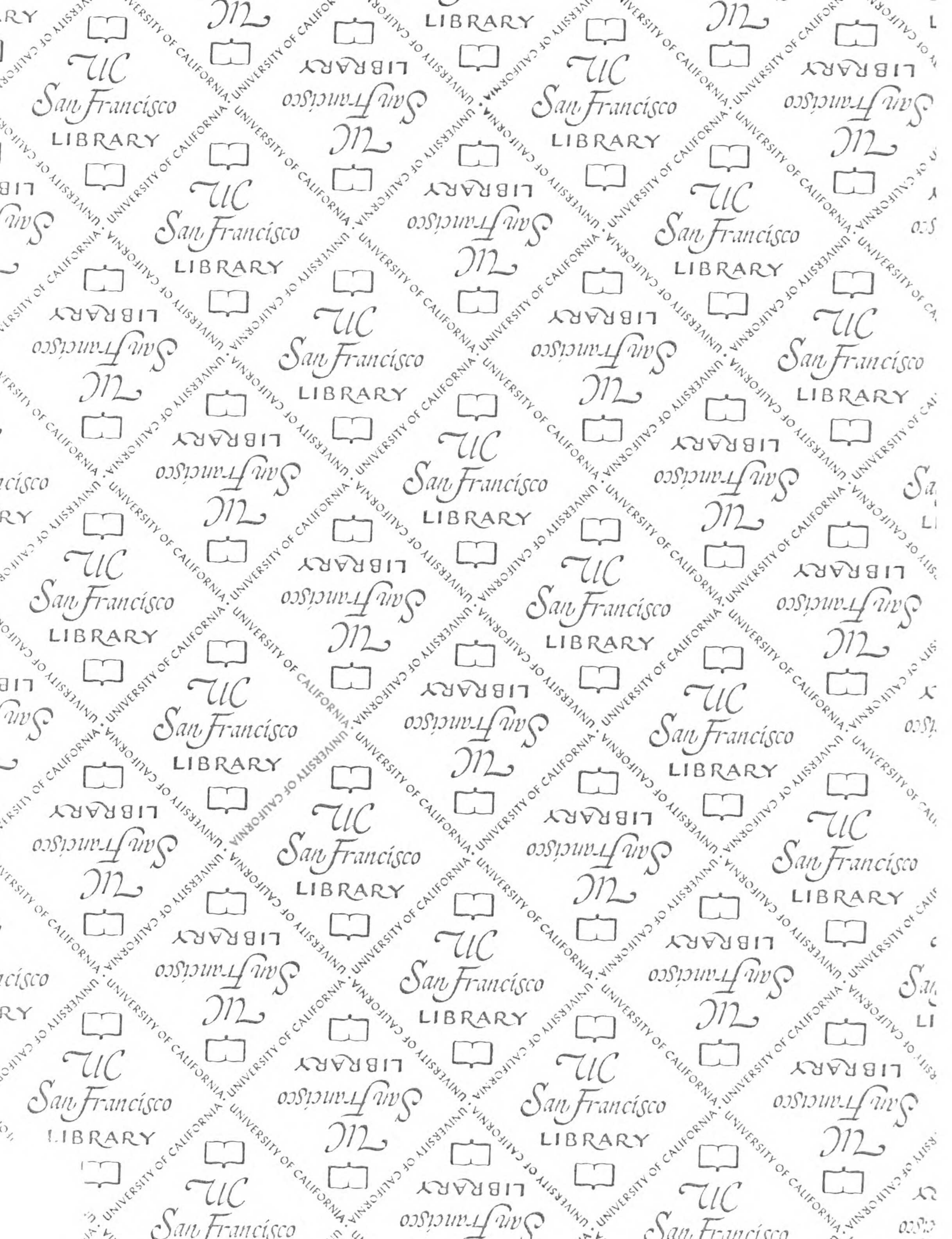
```
	state = dm__lock_state(tp->lock);
	if (state >= mode)
		return tp->key;

	command = (state == DM_NOLOCK) ? "LOCK" : "LOCKMODE";
	return get_tbl_lock(tp, command, mode, key, wait, TRUE);
}

/*
 * get_tbl_lock:
 *	Get a lock or change a lock mode
 */
static
int
get_tbl_lock(tp, command, mode, key, wait, new_level)
TABLE	*tp;
char	*command;
int	mode, key, wait;
int	new_level;
{
	char	buf[BUFLEN];

	(void) sprintf(buf, "%s TABLE \"%s\" %s %d %d\n", command, tp->name,
		(mode == DM_WRITELOCK) ? "write" : "read", key, wait);
	(void) write(fileno(tp->hp->fp), buf, strlen(buf));
	if (dm__resp_number(tp->hp->fp, &tp->key) < 0)
		return -1;
	(void) dm__lock_set(&tp->lock, mode, new_level);
	return tp->key;
}

/*
 * tbl_lock_change:
 *	Change the lock mode of a table
 */
int
tbl_lock_change(th, mode, key, wait)
DM_TABLE		th;
int			mode;
int			key;
int			wait;
{
	register TABLE	*tp;
	int		get_tbl_lock();

	tp = (TABLE *) th;
	switch (dm__lock_change(tp->lock, mode)) {
	  case LOCK_OKAY:
		return 0;
	  case LOCK_NEWMODE:
		break;
	  default:
		dm_errno = DM_ENOSUCH;
		return -1;
	}
	return get_tbl_lock(tp, "LOCKMODE", mode, key, wait, FALSE);
}

/*
 * tbl_unlock:
 *	Release the current level of lock
 */
int
tbl_unlock(th, key)
DM_TABLE		th;
int			key;
{
	register TABLE	*tp;
	int		unlock_table();

	tp = (TABLE *) th;
	if (key != DEFAULT_KEY && key != tp->key) {
		dm_errno = DM_EBADKEY;
		return -1;
	}
	switch (dm__lock_release(&tp->lock)) {
	  case LOCK_OKAY:
		return 0;
	  case LOCK_RELEASE:
		return unlock_table(tp, key);
	  case LOCK_NEWMODE:
		return get_tbl_lock(tp, "LOCKMODE", tp->lock->state, key,
				TRUE, FALSE);
	  default:
		dm_errno = DM_ENOSUCH;
		return -1;
	}
}

/*
 * unlock_table:
 *	Release lock on table
 */
static
```

libofs/table.c

```
int
unlock_table(tp, key)
TABLE	*tp;
int	key;
{
	char	buf[BUFLEN];

	(void) sprintf(buf, "UNLOCK TABLE \"%s\" %d\n", tp->name, key);
	(void) write(fileno(tp->hp->fp), buf, strlen(buf));
	return dm__check_reply(mmwb_resp_status(tp->hp->fp));
}

/*
 * attr_count:
 *	Count number of attributes of given type in a table
 */
int
attr_count(th, atype)
DM_TABLE		th;
int			atype;
{
	register TABLE	*tp;
	int		get_tbl_desc();

	tp = (TABLE *) th;
	/*
	 * Always get a new description since things may have
	 * changed since last we looked
	 */
	if (get_tbl_desc(tp) < 0)
		return -1;
	switch (atype) {
	  case DM_A_EXIST:
		return tp->nattr_exist;
	  case DM_A_FILLED:
		return tp->nattr_filled;
	  case DM_A_FULL:
		return tp->nattr_full;
	  default:
		dm_errno = DM_EBADARG;
		return -1;
	}
}

/*
 * attr_list:
 *	List attribute of a table
 */
int
attr_list(th, atype, names, n)
DM_TABLE		th;
int			atype;
char			**names;
int			n;
{
	TABLE	*tp;
	char	buf[BUFLEN];
	int	val;
	int	read_names();

	tp = (TABLE *) th;
	if (tbl_lock((DM_TABLE) tp, DM_READLOCK, DEFAULT_KEY, TRUE) < 0)
		return -1;
	(void) sprintf(buf, "LIST ATTRIBUTE \"%s\" %d\n", tp->name, atype);
	(void) write(fileno(tp->hp->fp), buf, strlen(buf));
	val = dm__read_list(tp->hp, read_names, (void *) names, (void *) &n);
	(void) tbl_unlock((DM_TABLE) tp, DEFAULT_KEY);
	return val;
}
```

```
			return -1;
		if (cell_write(ah, row, value) < 0) {
			(void) attr_close(ah);
			return -1;
		}
		return attr_close(ah);
}

/*
 * ofs_class_count:
 *		Count the number of object classes
 */
int
ofs_class_count(proj)
OFS_OBJECT	proj;
{
		OFS_OBJECT	oh;
		DM_TABLE	th;
		DM_ATTR		ah;
		register int	n;

		dm_errno = DM_ENOERROR;
		oh = obj_open(proj, "class", OBJ_CL_SIMPLE, OBJ_T_NONE);
		if (oh == NULL)
			return 0;
		if (prop_get(oh, "classes", (void *) &th) < 0) {
			(void) obj_close(oh);
			dm_errno = DM_ESYSERROR;
			return -1;
		}
		if ((ah = attr_open(th, "class")) == NULL) {
			(void) tbl_close(th);
			(void) obj_close(oh);
			dm_errno = DM_ESYSERROR;
			return -1;
		}
		n = cell_count(ah, DM_I_FILLED);
		(void) attr_close(ah);
		(void) tbl_close(th);
		(void) obj_close(oh);
		return n;
}

/*
 * ofs_class_list:
 *		List object classes
 */
int
ofs_class_list(proj, dbuf, n)
OFS_OBJECT	proj;			/* Generally the system project */
OFS_OBJ_CLASS	*dbuf;			/* Description array */
int		n;			/* Size of array */
{
		register int	count, i, nclass;
		OFS_OBJECT	oh;
		DM_TABLE	th;
		DM_ATTR		class, name, desc, owner;
		OFS_OBJ_CLASS	cbuf;
		char		buf[BUFLEN];
		short		sval;
#define	FAIL		{ dm_errno = DM_ESYSERROR; count = -1; goto done; }

		dm_errno = DM_ENOERROR;
		oh = obj_open(proj, "class", OBJ_CL_SIMPLE, OBJ_T_NONE);
		if (oh == NULL)
			return 0;
		if (prop_get(oh, "classes", (void *) &th) < 0) {
			(void) obj_close(oh);
			dm_errno = DM_ESYSERROR;
			return -1;
		}
		class = name = desc = owner = NULL;
		if ((class = attr_open(th, "class")) == NULL)
			FAIL;
		if ((name = attr_open(th, "name")) == NULL)
			FAIL;
		if ((desc = attr_open(th, "description")) == NULL)
			FAIL;
		if ((owner = attr_open(th, "owner")) == NULL)
			FAIL;

		/*
		 * Read in the classes now
		 */
		count = 0;
		nclass = cell_count(class, DM_I_MAX) + 1;
		for (i = 0; i < nclass; i++) {
			if (cell_read(class, i, (void *) &sval) < 0)
				continue;
			cbuf.class = sval;
			if (cell_read(name, i, (void *) buf) < 0)
				continue;
			cbuf.name = mmwb_cpystring(buf);
			if (cell_read(desc, i, (void *) buf) < 0)
```

```
			if (cell_read(desc, i, (void *) buf) < 0)
				continue;
			tbuf.description = mmwb_cpystring(buf);
			if (cell_read(owner, i, (void *) buf) < 0)
				continue;
			tbuf.owner = mmwb_cpystring(buf);
			if (count >= n) {
				dm_errno = DM_ETOOSMALL;
				count = -1;
				goto done;
			}
			dbuf[count++] = tbuf;
		}

		/*
		 * Clean up
		 */
done:
		if (type != NULL)
			(void) attr_close(type);
		if (class != NULL)
			(void) attr_close(class);
		if (name != NULL)
			(void) attr_close(name);
		if (desc != NULL)
			(void) attr_close(desc);
		if (owner != NULL)
			(void) attr_close(owner);
		(void) tbl_close(th);
		(void) obj_close(oh);
		return count;
}

/*
 * ofs_type_add:
 *	Add a new type
 */
int
ofs_type_add(proj, tp)
OFS_OBJECT	proj;
OFS_OBJ_TYPE	*tp;
{
	register int	row;
	OFS_OBJECT	oh;
	DM_TABLE	th;
	short		sval;
	int		put_cell();

	oh = obj_open(proj, "type", OBJ_CL_SIMPLE, OBJ_T_NONE);
	if (oh == NULL) {
		dm_errno = DM_EUNSUPPORTED;
		return -1;
	}
	if (prop_get(oh, "types", (void *) &th) < 0) {
		(void) obj_close(oh);
		dm_errno = DM_ESYSERROR;
		return -1;
	}
	row = inst_create(th);
	if (row < 0)
		return -1;
	sval = tp->type;
	if (put_cell(th, "type", row, (void *) &sval) < 0)
		return -1;
	sval = tp->class;
	if (put_cell(th, "class", row, (void *) &sval) < 0)
		return -1;
	if (put_cell(th, "name", row, (void *) tp->name) < 0)
		return -1;
	if (put_cell(th, "description", row, (void *) tp->description) < 0)
		return -1;
	if (put_cell(th, "owner", row, (void *) tp->owner) < 0)
		return -1;
	(void) tbl_close(th);
	(void) obj_close(oh);
	return 0;
}

/*
 * put_cell:
 *	Put in the value of a cell
 */
static
int
put_cell(th, aname, row, value)
DM_TABLE	th;
char		*aname;
int		row;
void		*value;
{
	DM_ATTRah;

	ah = attr_open(th, aname);
	if (ah == NULL)
```


UC
San Francisco
LIBRARY
UNIVERSITY OF CALIFORNIA

UNIVERSITY OF CALIFORNIA
UC San Francisco LIBRARY

FOR REFERENCE

NOT TO BE TAKEN FROM THE ROOM

CAT. NO. 23 012

PRINTED IN U.S.A.