The MIDAS database system

Thomas E. Ferrin, Conrad C. Huang, Laurie E. Jarvis and Robert Langridge

Computer Graphics Laboratory, Department of Pharmaceutical Chemistry, School of Pharmacy, University of California, San Francisco, California 94143-0446, USA

The Molecular Interactive Display and Simulation (MIDAS) database system is a hierarchical database specifically designed for complex macromolecular models, such as proteins and nucleic acids. Each molecular model consists of one or more molecules made up of a linear sequence of smaller units that we refer to loosely as "residues." Each of these residue units, in turn, is composed of one or more even smaller units called "atoms." The complete model might consist of a single component molecule, as in the case of a water molecule, or it might be a long chain of more complex residue components, such as in the case of amino acid and nucleic acid sequences. Complex functional groups, such as heme and NADH, can also be specified as single subunit components, and the user can then incorporate these groups into a larger model to form a single complex.

The various model component types are defined as arbitrary graphs of atoms with defined starting and ending points. The component type thus defines the connectivity of atoms for that component, as well as the linkage atoms to adjacent model components. Molecular "data" are stored in the "leaves" of the database hierarchy and are therefore directly associated with the atoms of a particular residue component, the component having been specified by type and position in the sequence of residues making up the chain. Individual atom data, however, are not restricted to a specific format or quantity, thereby allowing both flexibility and future extensions to easily be made to the database.

Keywords: molecular modeling, molecular databases, interactive graphics databases

Received 9 December 1987
Accepted 23 December 1987

BACKGROUND

As previously described, the MIDAS system was designed for real-time modeling of proteins and nucleic acids and their interactions with each other and with small molecules such as drugs (see "The MIDAS display system" on page 13). The design of a real-time molecular display system requires an underlying data structure that efficiently stores the large amounts of data associated with a macromolecular model and provides sufficiently fast access to support the real-time needs of the associated display program. Previous molecular modeling programs, such as CAAPS,[1] have combined the database access primitives within the basic program structure of the display program. This lack of program modularity is only a symptom, however, of a more fundamental database design deficiency within these early-generation molecular modeling programs. While existing relational database systems provide a convenient program interface, until recently these database management systems were limiting because of their relatively slow access time and large disk space requirements. Lack of an adequately performing database system for representing macromolecular data was the major motivation for designing our own database system.

THE NATURE OF MACROMOLECULAR DATA

The currently available macromolecular structure data* is largely the result of crystallographic studies of proteins and nucleic acids. These protein and nucleic acid biomolecules are built from smaller component molecules called residues and bases, respectively, and are chained linearly into large structures. (Although classically the term "residue" applies only to the amino acid subunit components of protein molecules, we use the term throughout this paper to indicate the generic repeating subunit component present in both proteins and nucleic acids.) The residues are typically less than 100 atoms in size and have a single linkage atom to the next residue in the linear sequence and a single linkage atom to the previous residue in the sequence. Additionally, in the case of proteins, a specialized linkage may exist between two nonadjacent residues in the sequence via a disulfide bond. A complex molecular model might consist of one or more chains of residues of varying length. Some chains might consist of only a single residue. For example, the hemoglobin molecule consists of four amino acid chains and four single-residue heme groups, totaling 578 residues and 4,556 (nonhydrogen) atoms.

The task of managing molecular models of this complexity from within a real-time modeling program would be nearly overwhelming even with present-day existing hardware† were it not that the diversity of biomolecules is due largely to residue sequence variations of only 20 common amino acids in proteins and four common nucleotides in nucleic acids. This redundancy in the structure of component residues of large biomolecules provides an efficiency in nature that we can use to advantage in our database design. Residue structure data such

*Published protein and nucleic acid conformational structure data is archived and distributed by the Brookhaven Protein Data Bank.

†The MIDAS database system was originally developed on a DEC PDP-11/70 computer for use with an Evans and Sutherland Picture System 2 graphics display.

as atom names, bonding pattern and linkage atoms need only be specified once and then applied to all like residues within a model. Only the actual coordinate data (3D position in a Cartesian coordinate system) is explicitly required for each atom. Using the example of hemoglobin again, there are only 20 unique residues among the total 578 residues. (The amino acid isoleucine does not appear in hemoglobin. Thus, there are 19 unique amino acids and the heme "residue.")

The nature of the macromolecular structure data thus lends itself to a hierarchical modeling approach. Each molecular model consists of one or more sequences of residues. Each residue, in turn, consists of atoms that are connected in a specific bonding pattern. Associated with each atom is the position of that atom in a Cartesian coordinate system. This hierarchy is summarized below:

- Model: consists of one or more chains of residues, each chain consisting of one or more residues.
- Residue: consists of one or more atoms. All atoms within a given residue are connected to form an arbitrary graph.
- Atoms: form the leaves of the hierarchy. Data associated with the individual atoms usually consists of atomic coordinates, but molecular surfaces or other arbitrary data can be stored as well. Each atom in the entire model can be uniquely described by a model number, residue sequence number and atom name.

Conceptually, this data model can be viewed as a "tree" of nodes (i.e., a hierarchy), where each leaf in the node contains coordinate data (see Figure 1). For attributes such as color, each leaf node can logically be considered as being subdivided into separate nodes for atomic bonds, labels and surfaces.

DESIGN GOALS

As previously described, a major goal in the design of MIDAS was to build a modeling system that processes the large amounts of data associated with macromolecules using little main memory, gives real-time performance with minimal CPU use and does not use excessive amounts of disk space for storing the molecular database. Clearly, the database subsystem is a critical component in achieving this goal. Other important goals related to the database subsystem include the desirability of storing both coordinate and noncoordinate data (e.g., atom and residue names), the ability to modify existing database records as well as to add entirely new records to an existing database (to facilitate, for example, the replacement of one amino acid residue with another),

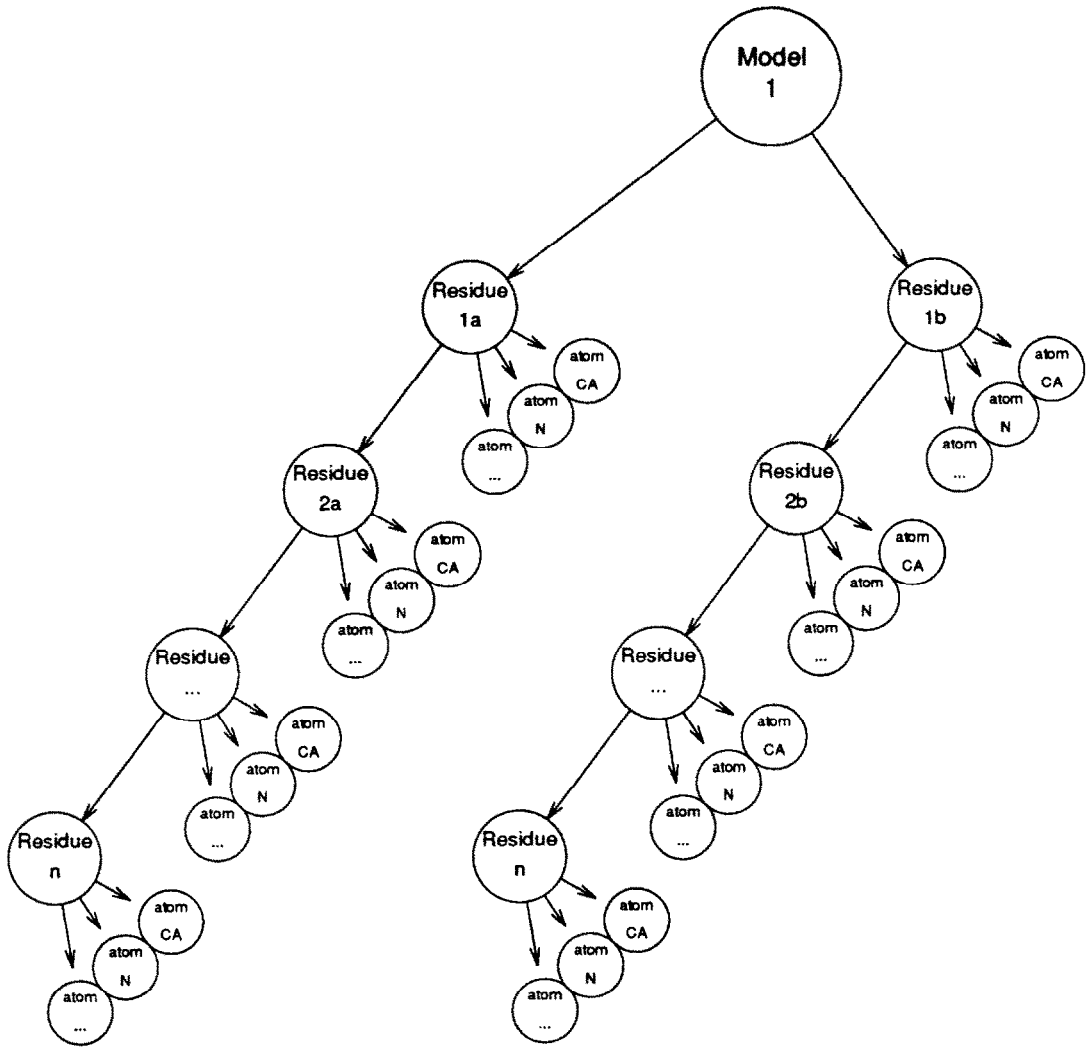

Figure 1. Conceptual organization of an MDBS Hierarchical database

the ability to do generalized "search" operations for data within a database, and, lastly, a means of extending the types of data stored within the database so as to be able to include the results derived from new ideas and discoveries. This last goal is particularly demanding, since it likely includes new types of data records and relationships that the designer did not originally consider; another way of stating this goal is to say that it is desirable to maintain an "open" database architecture.

DATABASE DESCRIPTION

First we'll describe the general file organization and design philosophy of the MIDAS database system (MDBS). Next we'll provide a brief summary of each of the user-level entry points into the MDBS package, followed by two sample programs that use the MDBS routines for creating a MIDAS database and then displaying the resulting model on a simple graphics display system. Appendix 1 provides a detailed description of the internal structure of the database records, while Appendix 2 provides a concise specification of the user interface and function of each routine.

All of the algorithms described here are programmed in C.[2] This language is suitable because the algorithms make extensive use of structures, pointers, recursion and dynamic memory allocation. Familiarity with the C language is assumed in the detailed description of database subroutine calls and in some of the more technical parts of the algorithm descriptions. Although we make no further reference to it, a Fortran-77 language interface to the MDBS subroutines also exists.

FILE ORGANIZATION

Each MIDAS database consists of three disk files: a template file, an index file and a data file (see Figure 2).

The template file consists of distinct entries for each different type of residue contained within the given molecular model. Each template record contains the residue type name, a unique name for each atom in the residue, and a description of the connectivity of atoms within the residue, including which atoms are the first and last in the template (i.e., those atoms that connect to the previous and next residues in the polypeptide chain, respectively).

As we will discuss below, the observant reader will note that template records are still "pinned"[3] in our data design, since an "index value" still points to particular records in the template file. However, this restriction does not prevent residues within existing databases from being substituted, or in some other way fundamentally modified, since the only possible consequence to the template file is having to create a new residue record at the end of the file. The only disadvantage of having pinned records in the template file is that the file cannot be kept sorted. Since the number of records within the file is always relatively small (in the range of 15 to 30), efficient linear searches can always be made.

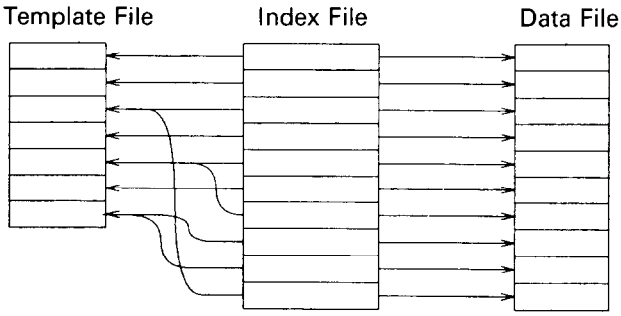

Figure 2. MDBS physical database organization

The fact that template files consist of only pinned records also does not prevent the concept of a "master" template file, provided the underlying operating system provides the necessary support for multiple links to the same data file (i.e., multiple directory entries referring to the same file system data). This means that, potentially, a single template file could be used for all protein and nucleic acid databases, since the total number of entries would be only of order 30 (20 amino acids plus 4 nucleotides plus a few special cases). Although the UNIX implementation of the MDBS subroutines use the shared template feature when saving away modified databases, this feature of the database design is not considered crucial to its success, chiefly because the data storage requirements for template files is already so minimal that the potential additional space savings is not significant. For example, the total storage requirements for all of our present 266 protein and nucleic acid data structures consists of 17.5 megabytes of disk space. Of this total, 0.7 Mb (4%) is used for storage of template files, 1.4 Mb (8%) is used for storage of index files (see below), and the remaining 15.4 Mb (88%) is used for storage of data files. We note in passing that the equivalent Protein Data Bank standard ASCII data representation requires 42.1 Mb of data storage; thus, the MDBS representation realizes nearly a 2.5 fold space savings in disk storage requirements.

The significant space savings possible with MDBS is also complemented by the far better data access times available when using a random access technique such as that provided by a "dense index."[3] In MDBS, the index file contains a table listing the ordering of residues within a molecule. This file serves as an index into both the template file (which describes the structure of each residue) and the data file (which contains the coordinate or other data associated with each residue). References to records in the template file are made via an 8-bit integer index to the particular residue structure record within the file, while references to the records in the data file are via a 32-bit direct pointer (equivalent to "file offset").

The existence of both the template file (which contains the residue name) and the index file (which contains the residue sequence number) provides for fast searches of the database. Searching by residue type proceeds linearly through a small memory-resident table (i.e., the template file), while searching by sequence number is done via a memory-resident binary search[4] of a somewhat larger table* (i.e., the index file); thus, a database "seek" operation always completes very quickly and without any accesses to disk.

*This can be done because the alphanumeric character strings representing sequence numbers are presorted lexicographically in the index file.

The date file contains only user-defined data associated with each residue. No structure is imposed on this data other than that it must be of fixed-length records and that the record size must be known. Users can store arbitrary data in this file. Because all data records are referenced with a direct pointer, all data can be accessed within a single disk seek. This provides for optimum data access and, in fact, makes MDBS viable even in today's typically rich virtual-memory environment. The fact that access to a particular virtual-memory page on secondary disk storage also takes a single disk seek as well means that the two access methods are roughly equivalent. The difference is that MDBS makes all the disk access decisions directly, while with virtual memory the operating system makes the disk access decisions. Since disk access time is at least three orders of magnitude greater than main memory access time, it makes little difference whether the program or the operating system makes the decision.

Records in the data file appear in random order, and thus disk blocks are completely filled. This is most important for MDBS data files, since they comprise approximately 90% of the disk storage requirements for any particular database (see above). New records are always added at the end of the data file, and deletions cause "holes" to appear in the file. Since deletions are an infrequent operation, this deficiency is not considered significant.

Appendix 1 includes more specific information on the internal structure of MDBS records. For most purposes, however, the subroutine package described here is sufficient to manage the database files, and potential users need not be concerned with the details of the internal file structure.

SUBROUTINE USE

The MDBS subroutine package provides a powerful and efficient means of accessing MIDAS databases. However, in order to effectively use the package, it is necessary to understand how stored information is accessed. In general, information can be held in three places:

(1) On disk: The MDBS disk files described above define the database and provide long-term storage.
(2) In internal buffers: Storage buffers in main memory are allocated and managed by the MDBS subroutine package when the database files are accessed.
(3) In user-defined buffers: MDBS subroutines return information the user requests into space the user allocates.

This trilevel data storage scheme allows a program to minimize the number of disk accesses for a given task. For example, searching the database for a specific residue determines if that residue exists, and this operation can be performed without actually reading any of the coordinate data associated with the residue. Likewise, it is possible to retrieve information such as the residue sequence number and type, and the number of atoms contained within the residue, but then access only the atom coordinate information if it is explicitly needed. For example, consider the steps for updating the coordinates for a single atom in a database record:

(1) Open the database for reading and writing.
(2) Locate the desired information, usually by "seeking" to a specific residue.
(3) Read all the atom data associated with the residue into main memory.
(4) Retrieve specific atom data from the internal MDBS buffer into user buffer space.
(5) Modify atom data and copy it back to the internal buffer.
(6) Write the residue data from main memory to disk when all modifications for a specific residue are complete.
(7) Close the database when all residue changes have been made.

Thus, database changes can be made on a convenient atom-by-atom basis, using only enough user-defined buffer space to hold a single atom's worth of data at any one time. This atom-by-atom access technique incurs little performance penalty because disk accesses are internally buffered on a residue-by-residue basis.

The routines available for manipulating databases can be divided into four categories, as shown in Table 1. Appendix 2 contains a detailed description of each MDBS user-level procedure.

EXAMPLES

Two simple examples demonstrate the ease with which the above described subroutines can be used. In these examples, some error-checking code and initialization routines have been omitted in order to retain clarity.

The first example involves converting a "Protein Data Bank" (PDB) format database* into a MIDAS format database. In this example, *readin()* is a routine that reads in data records from a PDB file and stores the retrieved residue type, residue sequence number and atom name as character strings and the atom coordinates as integers; the routine also returns the residue sequence number as an integer value. *Doinit()* and *done()* perform ancillary user processing for initialization and program cleanup prior to exiting.

```
#include <mdbs.h>

char    *midasfile;

main(argc, argv)
int     argc;
char    **argv;
{
        int     db, index, natom, nres, nprev;
        char    resseq[RES_SEQ_SIZE + 1], restype[RES_TYPE_
                SIZE + 1];
        char    atname[AT_NAME_SIZE + 1];
        int     coord[3];       /*space for atom coordinates*/

        doint(argc, argv);                      /*user initialization*/
        db = mopen(midasfile, "w");             /*open database for writing*/
        nprev = 0;
        /*read in atom records from PDB file*/
        while ((nres = readin(resseq, restype, atname, coord)) != EOF){
                if (nres != nprev){     /*if a new residue*/
                        if (nres != 1)  /*if not the first residue*/
                                mwrta(db);      /*write previous residue data to disk*/
                        /*now store residue information for new residue type*/
                        natom = mwrtr(db, resseq, restype, -1, sizeof(coord));
                        nprev = nres;
                }
                index = mseeka(db, atname);     /*find atom index*/
                mputa(db, index, coord);        /*save data in MDBS buffer*/
```

*PDB format was developed by the Brookhaven National Laboratory, which acts as a clearinghouse for macromolecular structure coordinate data.

Table 1. MDBS subroutine summary

(1)	Routines for opening and closing the database files:	
	mopen	Open a new or existing database
	mclose	Close a database
	msave	Save the current (and presumably modified) database
(2)	Routines to locate residue information:	
	mseekr	Seek to a given residue
	mrconn	Determine connectivity of two residues
(3)	Routines for passing information between disk and main memory:	
	mreadr	Read residue information into memory
	mreada	Read atom information into memory
	mwrtr	Write residue information
	mwrite	Write atom and residue information
	mwrta	Mark current residue as complete and advance to next residue
	mflush	Force information to be written out to disk
(4)	Routines for passing information between internal buffers and user buffers:	
	mseeka	Find an atom index
	matom	Find an atom name
	mlink	Return the index of the linkage (last) atom of a residue
	mchief	Return the index of the chief (first) atom of a residue
	mdatptr	Return a pointer to the atom data currently in main memory
	mgeta	Copy atom information to a user buffer
	mputa	Copy atom information from a user buffer
	mtrav	Traverse a residue connectivity graph
	maconn	Determine connectivity of two atoms
	mchain	Determine whether an atom is a mainchain or sidechain atom
	mamap	Return indices of all atoms connected to a given atom
In addition, three string comparison routines recognize MIDAS "wildcard" characters:		
	amatch	Atom name string comparison
	smatch	Sequence number string comparison
	tmatch	Atom type string comparison

```
        }
        mwrta(db);                  /*write last residue to disk*/
        mclose(db);                 /*close the database*/
        done();                     /*clean up and exit*/
}
```

The second example shows how the database created in the previous example can be used to draw a picture of the molecule on a simple graphics display system. Again, *doinit()* and *done()* perform ancillary processing, while *moveto()* and *lineto()* are subroutine calls to a graphics library package and are used to position the CRT beam and to draw a line, respectively.

```
#include <mdbs.h>

char    *midasfile;

main(argc, argv)
int     argc;
char    **argv;
{
        int     db, natom;
        char    resseq[RES_SEQ_SIZE+1], restype[RES_TYPE_SIZE+1];
        char    atname[AT_NAME_SIZE+1];
        int     visit(), again();

        doinit(argc, argv);     /*user initialization*/
        db = mopen(midasfile, "r") /*open an existing database for reading*/
        /*retrieve residue sequence number and type*/
        while ((natom = mreadr(db, resseq, restype)) > 0){
                mreada(db);     /*read atom coordinates into MDBS buffer*/
                mtrav(db, visit, again);    /*traverse the residue*/
        }
        mclose(db);
        done();
}

visit(db, index, ischief, islink, nsons, firstime)
int     db, index, ischief, islink, nsons, firstime;
{
        int     coord[3];
        static int firstatom = 1;

        mgeta(db, index, (char *)coord);        /*retrieve coordinate data*/
        if (firstatom){
                moveto(coord[0], coord[1], coord[2]);   /*graphics library call*/
                firstatom = 0;
        } else
                lineto(coord[0], coord[1], coord[2]);   /*graphics library call*/
}

again(db, index, ischief, islink, nsons)
int     db, index, ischief, islink, nsons;
{
        int     coord[3];

        mgeta(db, index, (char *)coord);        /*retrieve coordinate data*/
        moveto(coord[0], coord[1], coord[2]);   /*graphics library call*/
}
```

CONCLUSION

The MIDAS database system provides a flexible and compact data storage scheme for data associated with biomolecules. The access time to the data via the MDBS subroutine package is sufficiently fast to support a real-time graphics display system. The database access interface makes optimum use of available main memory to reduce disk accesses in the tasks of database search, storage and modification. The variety of database access subroutines offered allows the calling program a number of options in data access, main memory use and input/output operations that can be optimized for a particular application and programming environment.

REFERENCES

1 Langridge, R. Interactive three-dimensional computer graphics in molecular biology. *Computers in Life Science Research,* William Siler and Donald A. B. Lindberg, Eds., 1975, 53–59

2 Kernigham, B. W., and Ritchie, D. M. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 07632, 1978

3 Ullman, J. D. *Principles of Database Systems.* Computer Science Press, Potomac, MD 20854, 1980

4 Knuth, D. *Searching and Sorting, The Art of Computer Programming,* Vol. 3. Addison-Wesley, Reading, MA, 1972

5 Bernstein, F. C. The protein data bank: a computer archive. *J. Mol. Biol.,* 1977, **112,** 535–542

APPENDIX 1: DATA STRUCTURE DEFINITIONS

The MIDAS database system is organized conceptually as a hierarchical database. Each structure model consists of a linear sequence of residues; each residue, in turn, consists of a connected graph of atoms. Each atom forms the leaf of the hierarchy and has data associated with it. Usually, this data consists of atomic coordinates, but molecular surfaces or arbitrary user data can be stored as well.

The physical database for each model is arranged into three disk files: the *template* file (with suffix *.tpl*), the *index* file (suffix *.ndx*) and the *data* file (suffix *.dat*). An optional fourth file contains solvent accessible surface data (suffix *.srf*). Each file contains a leading header of the following format:

```
struct header__def {
        short     magic;
        short     rescnt;
        short     tplcnt;
        long__t   filesize;
};
```

The magic number field of this header structure is different for each of the files; this serves as both a file type identifier and a version specifier. For any specific molecule, the other three fields are always the same in each file. *Tplcnt* is the number of templates used in the model, and *rescnt* is the number of residues in the model. (This latter field is not actually used for template files because they can be shared by several databases.) *Filesize* is the size of the data file in bytes. These fields are not critical for accessing data in the file but are used for consistency checks.

The *template* file supplies the connectivity for each type of residue used in the model. Each template entry consists of a header and an atom list. The header has the following format:

```
struct tpl__def {
        short     natom;
        char      restype[RES__TYPE__SIZE];
        char      chief;
        char      linkage;
};
```

where *natom* is the number of atoms in the residue and *restype* is the ASCII name of the residue type (arbitrarily defined to be 6 characters maximum in length). *Chief* and *linkage* are indices of the first atom and "linkage" atom of the residue. The atom list consists of atom entries sorted in lexicographical order by atom name. Each atom entry has the format:

```
struct connec__def {
        char      cstat;
        char      tocnt;
        char      to[MAXTO];
        char      atname[AT__NAME__SIZE];
};
```

where *cstat* contains flags to indicate whether this atom is part of the molecule's "mainchain" backbone or whether the atom is part of a closed ring structure and hence cannot be part of a bond rotation. *Tocnt* is the number of atoms that are connected to the current atom (arbitrarily constrained to a maximum of 6), and *to* is an array of indices to these connected atoms. *Atname* is the ASCII name of the atom (up to 6 characters). The entire list of atoms makes up a unidirectional graph which when traversed in a depth-first search will provide an optimum graphical traversal; i.e., if the traversal were followed with a pen, the resulting trace would be a minimal traversal that still maintains proper connectivity.

The *index* file specifies how residues are connected within a complete molecule. In addition to the standard header, there is a second header of the following form:

```
struct reshdr__def {
        short     firstres;
        short     lastres;
        short     atomsize;
};
```

where *firstres* and *lastres* are indices to the first and last residues of the model. *Atomsize* is the size in bytes of the atom entries in the data file (24 bytes for each atom when using the standard MIDAS display format shown below). This second header is followed by the residue entries with the following structure:

```
struct residue__def {
        short     type;
        char      seq[RES__SEQ__SIZE];
        short     atomcnt;
        short     resprev
        short     resnext;
        long__t   offset;
};
```

where *type* is an index to the particular residue structure in the template file for this residue, and *seq* is the ASCII sequence name of the residue (again, arbitrarily set at 6 characters maximum). *Resprev* and *resnext* are the indices to the residues preceding and following the current residue, respectively. *Offset* is the address in the data file where the data for the atoms associated with the current residue are stored.

The *data* file consists of a list of atom entries. All atom entries for the same residue must be stored contiguously and be sorted by atom name. The datum size must be the same as that specified in the index file. Note that the number of atoms in the data file need not match the numbers of atoms in the template file, even if the residue type is the same. The data actually stored in this file is arbitrary. Thus, extensions to the

standard information usually supplied for molecules — such as atom coordinates, temperature factors and so on — can be made easily. The standard information supplied for all Protein Data Bank molecules has the following format:

```
struct  atom_def    {
        float   x, y, z;            /*atom coordinates*/
        float   tempfac;            /*temperature factor*/
        short   status;             /*atom status (e.g.,
                                    "visible")*/
        char    color, scolor, lcolor; /*bond, surface and label
                                    color*/
        char    pad1;               /*padding for longword
                                    alignment*/
        short   pad2;
};
```

The MIDAS database access routines are designed to be data independent; i.e., they assume no particular format for the data stored in the data file. Thus, the same utility routines can be used for different types of data, and application programs can then be made consistent throughout the system.

APPENDIX 2: DETAILED SUBROUTINE DESCRIPTIONS

The file */usr/include/mdbs.h* must be included in programs that use the MDBS subroutine package. If the programmer is accessing a database used by the MIDAS display program, the file */usr/include/midas.h* must be included as well. This latter file contains the structure definitions for the atom coordinate data and graphics display status and color data. A detailed description of each user-level MDBS subroutine follows:

1. Mopen

Calling protocol:

```
mopen(database,mode)
char*database,*mode;
```

where *database* is the name of the database to be opened, and *mode* is the mode with which it should be opened.

Mopen is used to open a database in a specified mode. The available modes are:

read	"r"	open an existing database for reading only
read/write	"rw"	open an existing database for reading and writing
write	"w"	create and open a database for writing only

Note that opening a database in "read" or "read/write" mode implies that the database already exists. Conversely, opening a database in "write" mode implies that the database does not exist and should be created. Creating a database that already exists truncates the original database. If an existing database is opened in "rw" mode (thereby indicating that the user intends to make changes), a temporary copy of the data file (*database*.dat) is made so that the original database will not be lost if the user subsequently decides not to save away changes after all. See the description of *msave* for additional information on saving modified databases.

Usually a database is accessed on a residue-by-residue basis. This means that data for an entire residue is stored in an internal buffer while updates and additions are made. The entire residue is written to the disk file upon completion of the updates. If large amounts of data are associated with each atom in a residue, however, the entire residue may not fit into main memory on a small computer. In this case, the database can be accessed on an atom-by-atom basis, designated "slow" mode.

In "slow" mode, each change to the database is written out to disk immediately instead of being stored in an internal buffer. As the name implies, this is a much slower method of access and should be employed only when the size of the atom data prohibits access in the default "residue-by-residue" mode. "Slow" mode is invoked by adding "s" to any of the above access modes. For example, "rws" indicates read/write in "slow" mode.

Mopen returns a small positive integer identifying the particular database "stream" through which the database can be referred to in subsequent operations. If *mopen* is unable to open the named database, a value of -1 is returned.

2. Mseekr

Calling protocol:

```
mseekr(stream,residue,mode)
char*residue;
```

where *stream* is the integer database identifier returned by *mopen*, *residue* is a characer string containing the name of a residue to be matched and *mode* is an integer bit mask whose value is defined in the table below.

Mseekr sets the position for the next input or output operation on *stream* by searching for a particular residue in the database. The search is merely a locating process and does not involve retrieving any information; it does, however, verify that a residue meeting the user's search criteria has been found. *Mseekr* returns the number of atoms contained in the given residue or -1 if the search fails.

The octal mask bits set in *mode* determine how the search is done:

Bit	Set	Reset
01	*residue* = type name	*residue* = sequence number
02	Search backward	Search forward
04	Search from logical start of model	Search from current position
010	Start of model = last residue	Start of model = first residue

Note that the starting position of the search is significant only if bit 04 is set and if *residue* is a type, since sequence numbers are unique within a database. If *residue* is a type, then the search proceeds to the logical end of database but does not wrap around. If bit 01 is set, then "*" can be used to match any residue type and "?" can be used to match any single character embedded in a string. To position *stream* at one end of the database, seek from the desired end for residue type "*". Searching by residue type proceeds linearly through a small

memory-resident table, while searching by sequence number is done via a memory-resident binary search of a somewhat larger table (this can be done because the alphanumeric character strings representing sequence numbers are presorted lexicographically in the index file); thus, the *mseekr* operation always completes very quickly and without any accesses to disk. Note that this call merely locates the desired residue and does not retrieve any data. Data is retrieved using the *mreadr* and *mreada* routines.

3. Amatch, Smatch and Tmatch

Calling protocol:

```
amatch(str1,str2)
smatch(str1,str2)
tmatch(str1,str2)
char*str1,*str2;
```

where *str1* and *str2* are character strings to be compared.

Amatch performs a string comparison of two atom names, *str1* and *str2*. *Smatch* performs a string comparison of two residue sequence numbers, *str1* and *str2*. *Tmatch* performs a string comparison of two atom types, *str1* and *str2*. The routines return zero if the strings match, nonzero if there is no match. The wildcard characters "*" (entire string match) and "?" (single character match) are recognized. Note that the "*" wildcard stands alone and cannot be used to match partial strings.

4. Mseeka

Calling protocol:

```
mseeka(stream,atomname)
char*atomname;
```

where *atomname* is the name of the atom to be matched.

Once the desired residue is located, *mseeka* can be used to find the index of a particular atom within the residue. This function is useful when the user knows only the name of the desired atom. For example, when creating a new database, it may be necessary to know the index of an atom in order to save the associated coordinate information with the *mputa* subroutine. *Mseeka* returns the specified atom index as an integer value. A −1 return value indicates that the atom does not exist.

5. Mchief and Mlink

Calling protocol:

```
mchief(stream)
mlink(stream)
```

where *stream* is the integer database identifier returned by a previous call to *mopen*. *Mchief* returns the index of the "first" atom in the current residue. This is the defined starting point (atom) for any residue, and it is this atom that forms a bond with the previous residue in a sequence of multiple residues.

Similarly, *mlink* returns the index of the "last" atom in the current residue. This is the linkage atom to the next residue in the sequence, if one exists.

6. Maconn and Mrconn

Calling protocol:

```
maconn(stream,index1,index2)
mrconn(stream,seq__num1,seq__num2)
char*seq__num1,*seq__num2;
```

where *index1* and *index2* are the indices of two atoms in the current residue denoted by *stream* and *seq__num1* and *seq__num2* are residue sequence numbers.

Maconn and *mrconn* are used to determine atom connectivity and residue connectivity, respectively. *Maconn* returns:

0	if the atoms are not connected
1	if *index1* precedes *index2*
2	if *index2* precedes *index1*
3	if *index1* and *index2* are the same atom
−1	if either atom index is invalid

Similarly, *mrconn* returns:

0	if the residues are not connected
1	if *seq__num1* precedes *seq__num2*
2	if *seq__num2* precedes *seq__num1*
3	if the residues are one and the same
−1	if either sequence number is invalid

7. Mchain

Calling protocol:

```
mchain(stream,index)
```

where *index* is an atom index in the current residue denoted by *stream*.

Mchain determines whether an atom is part of the mainchain of a molecule or, alternatively, part of a sidechain. *Mchain* returns:

0	if the atom is part of a sidechain
1	if the atom is part of the mainchain
−1	if an error occurs

8. Mamap

Calling protocol:

```
mamap(stream,index,indices)
int*indices;
```

where *index* is the atom index of an atom in the current residue denoted by *stream* and *indices* is a pointer to an array of integers.

Mamap returns the number of connections to atom *index* within the residue template and places the indices of these atom connections in the array pointed to by *indices*. Note that the indices are returned only for those atoms that reside in the template of the current residue. All connecting atoms are returned regardless of whether data is associated with the atom (this distinction is further elaborated upon in the next section). The array that receives the index values should be declared by

```
int  indices[MAXTO];
```

where *MAXTO* is defined in */usr/include/mdbs.h*.

9. Mreadr

Calling protocol:

```
mreadr(stream,resseq,restype)
char*resseq,*restype;
```

where *stream* indicates a previously opened database and *resseq* and *restype* are character arrays allocated by the user.

Mreadr retrieves the sequence name, type and number of atoms of the current residue. The name and type are stored in the user-supplied character arrays, and the return value of *mreadr* is the number of atoms in the residue. This call is useful for stepping through a molecule sequentially (perhaps to determine the residue sequence), but without reading the associated atom data.

10. Mreada

Calling protocol:

```
mreada(stream)
```

Mreada reads the atom data (typically atom coordinates, but potentially a molecular surface description or any other user-defined information) for the current residue from disk into an internal buffer. Note that the database *stream* must have been previously positioned to the appropriate residue. This call makes the atom data available to the user through subsequent calls to *mgeta* or *mdatptr*. Note that *mreadr* and *mreada* have distinct functions so the user can selectively retrieve data and avoid the expense of unnecessary disk accesses.

Mreada does nothing if the database was opened in "slow mode," since no internal buffering is done in this case.

11. Matom

Calling protocol:

```
matom(stream,index,atom_name)
char*atom_name;
```

where *index* is the index of the atom whose name is to be returned, and *atom_name* is the user buffer where the name is to be stored.

Matom retrieves the name of the atom specified by *index* for the residue denoted by *stream*. For example, if a user has made a call to *mlink* to find the index of the linkage atom, a call to *matom* then retrieves the name of that atom.

12. Mdatptr

Calling protocol:

```
char*mdatptr(stream)
```

Mdatptr returns a pointer to the internal MDBS buffer, which contains all the atom data for the residue denoted by *stream*. Note that the atom data must have been read into main memory already by a previous call to *mreada*. The *mdatptr* routine can be used to directly access atom data stored within an internal MDBS buffer, thereby avoiding the overhead associated with copying the information into user-allocated buffer space. *mdatptr* returns a pointer of type *char* because, in general, the datum size is not fixed. For MIDAS coordinate databases, however, the pointer returned will be of the type *struct *atom_def*, as defined in *midas.h*. See Appendix 1 for the definition of the atom data structure used by MIDAS.

MDBS atom entries are stored alphabetically by atom name. Thus, if a pointer returned by *mdatptr* is incremented, the atoms will be referenced in alphabetical order.

Mdatptr returns a NULL if the database was opened in "slow mode," since an internal buffer does not exist in this case.

13. Mgeta

Calling protocol:

```
mgeta(streatm,index,buffer)
char*buffer;
```

where *index* is the index of the atom whose information is to be copied and *buffer* is where the information will be stored.

Mgeta copies the information associated with atom *index* into the user-supplied buffer. The appropriate residue must have been read into main memory using *mreada* before the call to *mgeta*. If the database is being accessed in "slow" mode, the appropriate residue need only have been located (with an *mseekr* call, for example).

The user must ensure that the buffer provided is large enough to accommodate the data returned. For MIDAS coordinate databases, the space necessary is given by the C expression *sizeof(struct atom_def)*; however, for user-defined data structures, the buffer size will most likely be different.

14. Mtrav

Calling protocol:

```
mtrav(stream,visit,again)
int visit(),again();
```

where *visit* and *again* are functions to be called when each atom is visited and "returned to."

Mtrav is used to "traverse" through a residue. Since residues are defined as graphs with distinct starting and ending points, it is possible to trace a path through the atoms of the residue. Each time an atom is first reached via a new path, *visit* is called. Subsequent occurrences during traversal of this same atom as part of the same path will incur calls to *again*.

The user functions *visit* and *again* are called with the following arguments:

```
visit(stream,index,ischief,islinkage,nson,firstime);
again(stream,index,ischief,islinkage,nson);
```

where *index* is the index of the current atom. *Ischief* is an integer that is nonzero if the current atom is the first atom of the residue. *Islinkage* is nonzero if the current atom is the last atom of the residue. *Nson* is the number of atoms that are connected to the current atom and are yet to be visited. *Firstime* is nonzero if the atom is being visited for the very first time (it is possible for an atom to be visited twice via different paths if it is part of a ring).

Mtrav is most useful for generating instructions for

drawing a residue. To do this, *visit* would generate a *"lineto"* drawing command, and *again* would generate a *"moveto"* drawing command. (See the example in the main body of this article.)

15. Mwrtr

Calling protocol:

```
mwrtr(stream,resseq,restype,natom,atomsize)
char*resseq,*restype;
```

Mwrtr writes the residue information of the current residue on *stream* and returns the number of atoms in the residue. The residue sequence name and type name should be provided in *resseq* and *restype*, respectively. *Natom* is the number of atoms in the residue. If *natom* is negative, then the number of atoms is taken to be the number of atoms defined in the residue template. Thus, *mwrtr* can be used to determine the number of atoms in a residue. *Atomsize* is the size of an atom datum in bytes.

This routine must be called before *mputa* if the residue is being modified or created for the first time.

16. Mputa

Calling protocol:

```
mputa(stream,index,buffer)
char*buffer;
```

Mputa copies the data from atom *index* from the user space *buffer* into the current residue on *stream*.

17. Mwrite

Calling protocol:

```
mwrite(stream,resseq,restype,atoms,natom, atomsize)
char*resseq,*restype;
char*atoms;
```

Mwrite combines the functionality of *mwrtr* and *mputa*; it writes a residue whose sequence and type are *resseq* and *restype*, respectively. The associated atom data are found at memory locations beginning at *atoms*, and there are *natoms* pieces of data each of size *atomsize*. If *stream* is at the end of the database or *resseq* does not match the current residue sequence number, then the residue is appended to the end of the database. Otherwise, the current residue on *stream* will be replaced by the new residue.

18. Mwrta

Calling protocol:

```
mwrta(stream)
```

Mwrta sets a flag to indicate that all data for the current residue specified by *stream* has been stored into an internal MDBS buffer and that processing for the current residue is complete. The next call to *mreadr* then returns the residue information for the next residue in sequence. Making this call before a call to *msave* ensures that updating of the temporary database file is complete. This routine need not be called if *stream* is open in "slow mode," since in this case the data were already written to disk by a previous call to *mputa*.

19. Mflush

Calling protocol:

```
mflush(stream)
```

Mflush forces any data in the internal buffer associated with *stream* to be written out to disk. This routine is not normally called explicitly but is instead used internally by the MDBS database subroutines; its description is included here only for completeness. *Mflush* differs from *mwrta* in that the former always writes data to disk immediately, while the latter writes out data only when absolutely necessary.

20. Msave

Calling protocol:

```
msave(stream,newdb)
char*newdb;
```

where *newdb* is the name of the database into which the possibly changed data will be stored.

Msave copies the temporary database associated with *stream* to a new database named *newdb*. The temporary database must be open for both read and write access. If the new database name is omitted, then the original database is updated. The temporary database continues to remain open after the procedure call. Note that in order for the new database to contain all modifications, a call to *mwrta* must be made before *msave* is called. *Msave* will atempt to share template files between the original and new databases if possible. (See the section on file organization for additional details.)

Msave is useful for saving modified data without disturbing the parent database. Since all database changes are made initially to a temporary disk file rather than to the original disk file, *msave* functions the same for both default and "slow" access modes.

21. Mclose

Calling protocol:

```
mclose(stream)
```

Mclose is called to close an open database. If the database was opened for writing only, then the internal MDBS data buffers are flushed so that the database will reflect all updates. If the database was opened for both read and write access, changes are not implicitly saved; to save changes in this case, a call must first be made to the *msave* routine. *Mclose* also frees any previously allocated internal data buffers.

MISCELLANEOUS DETAILS

As described in an earlier section, the MDBS subroutines assume nothing about the format nor datum size of the user's data stored in the leaves of the database hierarchy. When preparing molecular models for display by the MIDAS display program, however, users must employ a specific atom data structure definition. These data definitions can be found in the file */usr/include/*

midas.h, and this definition file must be included in all programs that are preparing data for display by MIDAS. For typical atom coordinate data, the structure definition is as follows:

```
struct  atom_def {
        float   x,y,z;              /*atom coordinates*/
        float   tempfac;            /*temperature factor*/
        short   status;             /*atom status (e.g.,
                                    visible)*/
        char    color,scolor,lcolor; /*bond, surface and label
                                    color*/
        char    pad1;               /*padding for longword
                                    alignment*/
        short   pad2;
}
```

Status is a bit array of atom status information. Most of the atom status information is managed internally by the MIDAS display program and need not be referenced during database operations outside the scope of MIDAS. There are two exceptions, however. First, the EXISTBIT indicates that data actually exists for the given atom (i.e., that the coordinate data is valid), in contrast to the data record simply being a "place holder" in the database. Second, the STARTBIT indicates that the given atom starts a new chain (i.e., the bit is set for the first atom of the first residue in a residue sequence). These status bits are defined in */usr/include/midas.h* and can be set by:

```
buffer.status |= STARTBIT;
buffer.status |= EXISTBIT;
```

before calling *mwrta*.

Separate color information for atom bonds, surfaces and labels can optionally be specified by the user when a database is first created. Color value definitions are also defined in */usr/include/midas.h*.

Several years of experience have shown that the MDBS subroutines are easy to use and provide a "natural" way of referencing macromolecular models, particularly proteins and amino acids. Nevertheless, new users of the package often experience some difficulty when first trying to use the MDBS routines. Common errors include:

(1) As long as the database has not been opened in "slow mode" (as is the usual case), then data is not actually read into main memory until there has been a call to *mreada*. Thus, calls to *mgeta* without a call to *mreada* will return data that corresponds to the previously read residue (if any exists).
(2) As long as the database has not been opened in "slow mode," then data is not marked for output until a call to *mwrta* has been made.
(3) When traversing a residue using *mtrav*, an atom can be visited twice if it is part of a ring. The *firstime* variable indicates whether the visit is actually the very first one.
(4) In order to traverse residues using *mtrav*, all atoms must be present, whether there is valid data associated with them or not. This means the user may have to keep a status word along with the data in order to differentiate between real data and space fillers (this is the function of the EXISTBIT noted above).
(5) When a database is opened for both reading and writing, the updates made to that database are not automatically saved on closing. A call must be made to *msave* in order to save the changes.
(6) *Mclose* must be called in order for temporary files to be deleted. Programs that abort abnormally without calling *mclose* will leave temporary files on disk (this causes no serious consequences).