DCDB database engine

by David F. May

May 10, 2006

Contents

1 Introduction

This is DCDB (David's C DataBase engine). It is a database engine designed around the Indexed Sequential Access Method (ISAM). A valid question at this point could be, "Why another database application for Linux?". There are numerous projects that are open source and that meet the needs of various segments of Linux users. However, there is a hole in the area of database engines that are open source and that allow functionality similar in scope to CodeBase (TM). The following are some things about the DCDB database engine that set it apart from other database products.

1. DCDB is not relational. There is no built in functionality to maintain relations between table items. If you need such functionality for your applications, you may want to consider Typhoon or some other such open source project.

2. DCDB is not an SQL database. If you need a good SQL database, you would be better served by one of the excellent open source SQL database projects currently available, like MySQL or PostgreSQL.

3. DCDB is not an xBase database engine. There is no attempt in DCDB to support the xBase file format, which in my estimation is a limiting format. The file format of DCDB tables and indexes is totally unique to DCDB, as far as I know.

4. DCDB is not very feature rich, although it does have features that have made it useful to me in my needs for database functionality. It is fast and has been tested extensively under conditions on all the supported platforms.

5. DCDB is open source. Other database engines, like C-ISAM or CodeBase are large and feature rich. They are documented well and provide an excellent value if you can afford them. However, they are expensive, even more so if they support multiple platforms. DCDB is designed to be totally free from the ground up.

6. DCDB is designed to be embedded into scripting languages, like TCL/Tk or Perl. That way, you can take a scripting language that you are somewhat familiar with, tack on the DCDB bindings and have a simple database component to the language. This gives you the ability to manipulate tables and indexes from a language you know using a fairly fast database engine.

The goals of the DCDB project are as follows (in order of priority):

1. Stability.

   Priority number one is to insure stability in DCDB. Bug fixes will take priority over any feature enhancements. Debugging directives that allow us to check for memory leaks, dangling pointer problems, etc. have been coded right into the system at all levels to help the programmer insure stability in final applications. The error checking and reporting at all levels of the code is extensive (if not exhaustive). Also, the base functionality of the code has been tested extensively. The tests were coded and implemented at the time that the various features were implemented, so that further features could be guaranteed to not produce problems with current functionality. Finally, the table maintenance code is designed to store data relatively safely without compromising performance.

2. Performance.

   After stability, performance is of top priority. I want a database engine that is fast and streamlined. The fat has been trimmed from as much of the base functionality as possible. DCDB is built upon very little in the way of underlying code; a handful of modules make up the base functionality upon which DCDB is built. Various parameters (the number of cached blocks in indexes, for example) are determined based on extensive testing. Also, various sorting algorithms were tested to determine which would be preferable for indexing. The indexing algorithm uses a split index file (in-memory indexes to blocks of index information). This is sort of a combination of a hash and B+tree, giving you the performance of a hash with the advantage of sequential, ordered traversal. It was implemented for performance.

3. Well documented.

   Open source projects tend to be difficult to use because of poor documentation. With the DCDB project, it is hoped that this trend can be reversed somewhat. I aim to carefully document in the code itself the

features of the database engine. Also, I aim to provide samples that are well documented and that allow the end user to get some feel for how to use DCDB. These will be fairly simple applications that are instructive in scope. This should minimize the time that it takes a user to get up and working with DCDB.

4. Future goals...

   I would like to make DCDB multi-threaded at some point. Also, I should probably add a locking mechanism that is a little more advanced than file locking (although this has worked well with serial programs, etc.). I think the capability to open a table and index in read-only mode could be useful for certain applications (like a database on a CD-ROM). Finally, I need to redesign the index to provide for reasonable record update capability.

2 DCDB constants and types

There are numerous constants that determine how DCDB works. Most of these are arbitrary values that can be changed by the user to enhance the functionality of DCDB (like MAX_IDX, which is the maximum number of indexes for a table). Others are based on testing and should be confirmed through experimentation before changed (like INDEX_BLOCK_SIZE, the default index block size). Those that were derived from testing have "tested" in a comment field beside them.

2.1 Error handling

Errors are communicated to the user in DCDB by way of a few global variables and some error handling information in the table structure. However, error handling is simplified by a few macros that decay to tests of the error variables for exception conditions. The constants used in error handling in the DCDB routines are as follows:

```
enum __dberror_type {
    DB_NOERROR,         /* no problema */
    DB_IO_DENIED,       /* I/O Error - FIO_DENIED */
    DB_IO_TOOMANY,      /* I/O Error - FIO_TOOMANY */
    DB_IO_NOFILE,       /* I/O Error - FIO_NOFILE */
    DB_IO_BADFD,        /* I/O Error - FIO_BADFD */
    DB_IO_READWRITE,    /* read or write not complete */
    DB_IO_PROHIB,       /* I/O Error - FIO_PROHIB */
    DB_IO_DEADLK,       /* I/O Error - FIO_DEADLK */
    DB_IO_NOLOCK,       /* I/O Error - FIO_NOLOCK */
    DB_NOMEMORY,        /* memory Error */
    DB_INVALIDFILE,     /* invalid cdb table or index */
    DB_LISTERROR,       /* error from the list level */
    DB_FIELD,           /* field error */
    DB_INDEX,           /* index error */
    DB_CREATEINDEX,     /* couldn't create the index */
```

```
    DB_UNSTABLE,        /* memory elements are corrupted */
    DB_UNIQUE,          /* unique index constraint violated */
    DB_TOOMANYIDX,      /* too many indexes for this table */
    DB_PARAM,           /* bad function parameter */
    DB_SHELL,           /* error from the shell level */
    DB_LOCK,            /* table locked */
    DB_PARSE,           /* error from the parser */
    DB_SYNTAX,          /* syntax error */
    DB_INPUT,           /* bad input */
    /* put new ones here */
    DB_UNSPECIFIED      /* unspecified error */
};

typedef enum __dberror_type dbErrorType;

extern dbErrorType dbError;

extern char *dbErrMsg[];    /* error messages */

#define isTableError(t) (!(t->dbError==DB_NOERROR&&dbError==DB_NOERROR))
#define isDBError()     (dbError != DB_NOERROR)
```

Errors can occur in two contexts in the DCDB routines. There are errors which occur during the creation, opening or closing of a table and during use of a workspace. These areas are considered to be global in scope (not table specific) because we don't have a valid table descriptor to attach the error to. However, other errors in the DCDB routines are table specific and therefore can and should be attached to the table they occurred for. In that case, the table descriptor will have members that show error information.

If the error occurs in a global context, the error will be kept in "dbError". The message pertaining to the error would be available in "dbErrMsg[dbError]". If, however, the error is table specific, the error will be kept in "tbl->dbError" and "dbErrMsg[tbl->dbError]" will point to a message describing the error.

The problem here is that this is not easy to remember. Therefore global errors can be checked by using the macro "isDBError()". This will be TRUE if an error has occurred in a global context. A description of the error can be gotten by calling the function "dberror()". If the error is table specific, the macro "isTableError(tbl)" will be TRUE and the user can get a description of the error with a call to "dbtblerror()". You should be aware that if an error occurs, any further calls to DCDB functions will not work correctly, as all DCDB routines do consistent error checks during processing. Therefore, if an error occurs, you *must* call dberror() or dbtblerror(), depending on the context, to clear the error condition before you can call DCDB routines to close tables and indexes by way of cleanup.

2.2 DCDB types

The basic structure of the cdb module is the dbTable structure. This is the data type that facilitates the management of data in the table. The first part of this structure is a header, which is saved to the table on disk directly as is. This header is of type dbHeader, which is defined in cdb.h. The second member of the dbTable structure is an array of field structures of type dbField. These are kept in the table on disk as well. The rest of the fields in the dbTable structure are used for managing a table in memory.

The format on disk of a cdb table is as follows:

```
|--------|--------|--------|------ ... |--------|---- ...
header   field 1  field 2  field 3 ... field n  records...
```

There are no special delimiters between parts of the table. The first 4 bytes of the table will be a magic number, #defined as DB_MAGIC in cdb.h.

```
/*
 * Header information.  This is the first thing stored in a
 * data file.
 */
#define DB_MAGIC    0xbeeecaafUL    /* be careful */

struct __db_header {
    unsigned long magic;               /* magic number - */
                                           /*   denotes one of ours */
    char timeStamp[TIME_STAMP_WIDTH+1];/* date and time */
                                           /*   of last update */
    char tableInfo[TABLE_INFO_WIDTH+1];/* information */
                                           /*   specific to the */
                                           /*   application */
    char midxInfo[MAX_MIDX][TABLE_INFO_WIDTH*2+1];
                                           /* multi-field index information */
    int nextSequence;                  /* next sequence number */
    int numFields;                     /* number of fields in the table */
    unsigned long numRecords;          /* num of records in the table */
                                           /*   currently */
    size_t sizeFields;                 /* size of field descriptions */
                                           /*   in the table */
    size_t sizeRecord;                 /* size of each record */
};

typedef struct __db_header dbHeader;
```

The “timeStamp” member is just the result from a call to asctime(). The “tableInfo” member is anything the application wants it to be. The “midxInfo” member is how the DCDB library keeps track of multiple index information. The “nextSequence” member is provided to allow users to have a sequence for

any given table. This is stored in the header so the management routines can make sure that a unique number is provided to the user each time. For example, if your application requires a unique purchase order number for each document it tracks, you could initialize tbl->nextSequence to 1. Then, when a document is created, you could use this to derive a purchase order number (like P00000001) and then increment the next sequence. This gives you the ability to create a unique number for each purchase order and makes sorting by purchase order number reasonable. The nextSequence information is stored in the table header, so it will be that same value after you close the table and reopen it.

The "numFields" member will contain the number of fields in the table. The "numRecords" member is the number of records the table currently contains. The "sizeFields" member contains the size of all the field headers in the table combined. So, the size of the dbHeader plus the sizeFields member will give the user the offset in the file where records are stored. The "sizeRecord" member will contain the size of each record in the table. The DCDB module stores records in a table in constant width fields that are padded with 0's (binary 0's, not '0's). So if we move to the first record in the table and retrieve sizeRecord bytes from it, we will get all the data from the first record. The first byte of the record is a character indicator of whether the record is deleted or not.

The dbField structure is defined as follows:

```
typedef enum _index_type {
    ITYPE_NOINDEX,          /* no index */
    ITYPE_UNIQUECASE,       /* unique index, case */
    ITYPE_UNIQUENOCASE,     /* unique index, no case */
    ITYPE_DUPCASE,          /* dups allowed, case */
    ITYPE_DUPNOCASE,        /* dups allowed, no case */
    /* Add new ones here */
    ITYPE_LAST
}   indexType;

typedef enum _field_type {
    FTYPE_NONE,             /* no field type */
    FTYPE_CHAR,             /* character */
    FTYPE_NUMBER,           /* number */
    FTYPE_LOGICAL,          /* logical */
    FTYPE_DATE,             /* date */
    FTYPE_TIME,             /* time stamp */
    /* Add new ones here */
    FTYPE_LAST
}   fieldType;

/*
 * Field information.  One of these is stored for each field.
 */
struct __field_descriptor {
    char fieldName[MAX_FIELD_WIDTH+1];/* name of field */
    char indexName[INDEX_NAME_WIDTH+1]; /* name of */
```

```
                            /*   index file for */
                            /*   this field */
    fieldType ftype;        /* type of field */
    indexType indexed;      /* type of index */
    int indexBlkSize;       /* block size of the index */
    int validated;          /* TRUE => validated */
    int fieldLength;        /* total length of the field */
    int decLength;          /* length of the decimal part, */
                            /* including the */
                            /*  decimal (if applicable) */
};

typedef struct __field_descriptor dbField;
```

The "fieldName" member is the name of the field. The "fieldType" member is one of character string, floating point numerical data, logical/boolean value (can be one of "tTyYfFnN"), date (an 8 digit date kept in YYYYMMDD format), and a time stamp as returned by asctime(). Note: See "indexType" above for the actual constants used. There is no support for a memo field. If the user requires storage of textual information, a more portable way is to create a table to save the text a line at a time. The "indexed" member will indicate the type of index applied to this field, if applicable. You have five possibilities with indexes. The item may not be indexed. If it is, DCDB supports unique indexes (as well as non-unique indexes) and case sensitive/insensitive indexes. The possibilities are as follows:

- no index (which means this field isn't indexed)
- unique, case sensitive index
- unique, no case index
- duplicates allowed, case sensitive index,
- duplicates, case insensitive index.

Note: See "fieldType" above for the actual constants used.

There is also support for multiple field indexes. These are indexes which contain sorted information for 2 or more fields. The field items are concatenated together and indexed. The information for multiple fields is kept in the table header instead of field records.

The "validated" field indicates whether validation should be applied to this field before data is entered to the table. This is currently not supported; however, it may be. The idea is that you can insure that a field only gets certain pre-defined values. For example, if you have a field for tracking part numbers in a table, you would want to insure that only certain part numbers are entered in that table. Therefore, you would set up another table to validate this part number field and the system wouldn't allow any but valid data to be entered.

"fieldLength" is the total length of this field (not including the nul byte). "decLength" is the length of the decimal part, if applicable. For all but NUMBER fields, this will be 0.

The dbTable structure is declared as follows in cdb.h:

```
struct __db_table {
    int fd;             /* file descriptor for table */
    dbHeader *hdr;          /* header information */
    dbField **fldAry;       /* array of field descriptors */
    char **fieldNames;      /* names of the fields in the */
                    /*   table */
    char *fileName;         /* name of file table is in */
    off_t offset;           /* current offset into the */
                    /*   file from 0 */
    off_t crec;         /* current record */
    dbErrorType dbError;        /* error information */
    char *dbErrMsg;         /* error message for the */
                    /* current error */
    int bof;            /* at beginning? */
    int eof;            /* at end? */
    char *data;         /* storage for field entry on */
                    /*   retrieve */
    char **fields;          /* pointers to storage for */
                    /*   entering */
                    /* a field */
    int *flens;         /* lengths of each field */
    int *declens;           /* decimal length for each */
                    /* field, including the decimal. */
    dbIndex *current;       /* current index */
    ListHeader *idxList;        /* index list */
    ListHeader *midxList;       /* multi-field index list */
    /* the following are currently not supported */
    dbCondType *ctype;      /* condition type for queries */
    ListHeader *entryCache;     /* cache for entries */
    ListHeader *searchCache;    /* cache for searches */
    ListHeader *queryCache;     /* cache for queries */
    char *betweenTop;       /* high value for a between */
                    /*   clause */
    ListHeader *inList;     /* list for "in" clause */
};

typedef struct __db_table dbTable;  /* table type */
```

The "fd" member contains the file descriptor for the open table. "hdr" is the table header (type dbHeader) and "fldAry" is an array of fields (type dbField) with a NULL pointer at the end of the array. "fieldNames" is an array of character pointers that will point to the names of the fields in "fldAry". These should be considered read-only; they point to the fieldName member of each item in "fldAry". The "fileName" member is the name of the file on disk.

The "offset" member will contain the offset in bytes of the start of the current record and "crec" contains the current record number less one. "dbError" will contain the error code in the event of an error (see error handling above). "bof" indicates the table pointer is at the beginning of the file and "eof" indicates it is at the end. These are used in sequential access of the table; if the user goes off of the beginning or end of the file, bof or eof is set, respectively.

The "data" member is storage the size of a record on disk, for moving data between memory and disk. "fields" is an array of character pointers with enough storage for each field. "flens" is an array of integers that corresponds to the fields array. The size given by "flens" should be seen as the maximum size of string to copy to the fields item, so that

```
strlen (tbl->fields[i]) <= tbl->flens[i]
```

"current" is the current index, if there is a current index, for the table. "idxList" is a ListHeader for the index list if there are indexes open for this table. "midxList" is a list that holds the multi-field indexes for the table that are currently open, if there are any.

Support for workspaces has been added to the DCDB library to facilitate the grouping of tables together in a logical way. The user can create the tables that are all part of a single application, for example, set default current indexes for each table, and add them to the workspace. Then, whenever a workspace is closed, all the tables that are managed by the workspace are closed. Whenever it is opened, all the tables that are grouped together by it are opened. This is just a convenience for the user. The following structures are used to manage workspaces in the DCDB library:

```
/*
 * Workspace structures
 */
typedef struct _ws_header {
    unsigned long magic;                        /* magic number */
    char wsName[TABLE_INFO_WIDTH+1];    /* WS name */
    int numWS;                          /* number of tables */
}   wsHeader;

typedef struct _ws_field {
    int fieldNum;
    dbTable *tbl;
    char tableComment[TABLE_INFO_WIDTH+1];
    char tableName[TABLE_NAME_WIDTH+1];
    char currentIndex[INDEX_NAME_WIDTH+1];
}   wsField;

typedef struct _work_space {
    wsHeader *hdr;
    int number;
```

```
    char fileName[TABLE_NAME_WIDTH+1];
    shellHeader *fields;
    wsField *current;
}   workSpace;
```

The "wsHeader" type contains the data that is stored to disk in the first bytes of a workspace. The "magic" member uniquely identifies this as a DCDB file. The "wsName" member is the name of the workspace. The "numWS" member is the number of tables grouped together by this workspace.

The "wsField" structure is used to maintain the table information in a workspace. One member of this type is stored to disk for each table grouped together by the workspace. These are also maintained in memory for table management. The "fieldNum" member is the number of this wsField. When tables are added to a workspace, a wsFeild is created and given the next number in the sequence. This is how they are stored in memory and on disk. The "tbl" member is a pointer to the table descriptor. This is only used with the field in memory. The "tableComment" is application dependent. It is there to give the user the opportunity to store application specific information about each table. The "tableName" member is the name of the table for this field. The "currentIndex" is the default index that is set to current when the table is opened. This is determined by the workspace routines when the table is added to the workspace.

The "workSpace" structure is the actual type that the user will work with. "hdr" is the wsHeader for this workspace. "number" is the number of tables stored in this workspace (same as "hdr->numWS"). "fileName" is the name of the file the workspace is stored in (same as "hdr->wsName"). "fields" is a sorted shell of the wsFeild types for all the tables. "current" is the table currently in use by the workspace routines.

3 DCDB Table Functions

This file includes the routines to create, open and close a table, as well as the routine to save the table header. The process that a user should follow to create a table is this:

- create the table (see buildTable()).
- create all the indexes (see createDBIndex() and createMultiIndex()).
- close the table and indexes (closeTable()).
- reopen the table and indexes (openTable()).

The valid table pointer returned by openTable() can then be used for adding, updating, and searching table data.

3.1 createTable

```
dbTable * createTable (const char *fname, dbHeader * hdr,
             dbField * flds[])
```

createTable() creates a cdb table. "fname" is the name of the table to create. "hdr" is a pointer to a dbHeader type that should have the tableInfo field set, either to all binary 0's or what the application needs.

The "flds" parameter is an array of pointers to dbField types. The last item in this list should be NULL. Every member of the dbField structure in each item pointed to by the "flds[]" array should be filled. The dbFields records pointed to by the items in the "flds[]" array are saved to disk unchanged, so it is contingent on the caller to insure that the fields are set up correctly.

Upon successful completion, createTable() will have created a DCDB table and returns a valid pointer to a dbTable object. This object is allocated on the heap and should be freed before the program exits by calling closeTable(). NULL is returned by createTable() on error. The user can use the isDBError() macro to determine if an error has occurred and the function dberror() to communicate the error to the user.

The "hdr" and "flds[]" parameters should be allocated on the heap and are assumed to belong to the table structure on successful completion of this call. When you call closeTable(), it will attempt to deallocate the storage with free().

Because of the low-level nature of the createTable() function, it should not be used by the end user to create tables. The buildTable() function should be used instead.

3.2 openTable

```
dbTable * openTable (const char *fname)
```

openTable() opens a table that already exists. "fname" is the name of the table on the file system. A valid pointer to a dbTable structure is returned upon success, NULL is returned otherwise. All indexes for the table are opened when the table is opened and are managed by the table members idxList and midxList.

If an error occurs, isDBError() will be true and dberror() will return a string that describes the error.

3.3 openTableNoIndexes

```
dbTable * openTableNoIndexes (const char *fname)
```

openTableNoIndexes() opens the table with the name "fname" without opening the indexes associated with the table. This is useful for applications in which tables have corrupted indexes and the application still has to open the tables.

Upon success, openTableNoIndexes() returns a valid table descriptor. If an error occurs, isDBError() will be TRUE and dberror() will return a description of the error that occurred.

3.4 closeTable

```
int closeTable (dbTable * tbl)
```

closeTable() closes all open indexes associated with the table, writes the table header information, flushes the table file, closes the table and frees all the dynamically allocated memory "owned" by the table structure. Because of the caching that occurs with table files and index files, it is very important that the user close all open tables before the application exits. If a table is not closed, it is very likely that data corruption will occur. If the indexes are not closed, it is almost certain that index corruption will occur.

closetable() returns _OK_ on success. If an error occurs, isDBError() is TRUE and dberror() will return a string describing the error which occurred. closeTable() will make every attempt to continue closing a table even when errors occur, to insure that data is preserved if possible. Therefore, if multiple errors occur, the error indicated by dberror() will be the last one to have occurred.

3.5 storeTableHeader

```
int storeTableHeader (dbTable * tbl)
```

storeTableHeader() stores the table header. This stores the table header to disk. This is done before the table is closed to insure that the memory data gets flushed to disk, but it can also be done by the user at any time during execution. The time stamp is updated and the header is copied to the beginning of the file.

The table header is stored to disk if this function is successful. If not, isTableError(tbl) will be TRUE and dbtblerror() will return a string that describes the error.

3.6 storeFieldHeader

```
int storeFieldHeader (int num, dbTable * tbl)
```

storeFieldHeader() stores the "num" field header of table "tbl" . "num" is assumed to hold the field number (0-based, so 0 => first field, etc.). "num" should be passed to the function such that tbl->fldAry[num] == field to be updated. If storeFieldHeader() is successful, it returns _OK_. If not, isTableError(tbl) will be TRUE for the table and dbtblerror() will return a description of the error.

4 DCDB Index Functions

This file provides the index support to the cdb table functions. It includes the functionality to create, open, close, query and search on indexes.

4.1 createDBIndex

```
int createDBIndex (dbTable * tbl, char *idxName, char *fldName,
           int isCase, int isUnique, int blkSize)
```

createDBIndex() creates an index file for the table "tbl". "idxName" is the name of the index. There will be two files created for each index, one with the extension ".inx" and one with the extension ".idx". The ".inx" file is the memory index and the ".idx" file is the block file for the index. The names of these files will be "idxName" concatenated with ".inx" or ".idx", respectively. "fldname" is the name of the field to create the index for. If fldname is NULL or does not contain a valid field (one that matches at least partially one in the field list), an error is indicated. "isCase" is TRUE if the index should use case sensitive sorts, FALSE otherwise. "isUnique" is TRUE if the index should not allow duplicates, FALSE if duplicates are allowed. "blkSize" will be the block size of the block file the index is stored in. If "blkSize" is 0, INDEX_BLOCK_SIZE (defined in cdb.h) will be used. The block size for an index file should be a power of two such that 128 <= block size <= 4096. A good rule of thumb (where performance is concerned) is to use a block size such that there are between 20 and 35 items stored per block. More items than this per block and performance tends to degrade. However, it is recommended that the block size be chosen based on performance tests for the items that are going to be indexed.

If successful, createDBIndex() creates the block file for the index, sets the current index to the newly created index and returns _OK_. If an error occurs, isTableError(tbl) for that table will be TRUE and dbtblerror() will return a description of the error.

4.2 createMultiIndex

```
int createMultiIndex (dbTable * tbl, char *midxName, char **names,
          int isCase, int blkSize)
```

createMultiIndex() creates a multifield index for the table given by "tbl". "midxName" is the name of the index (the naming of multi indexes is the same as it is for field indexes in createDBIndexes()). "names" is a NULL terminated array of character strings that indicates which fields are represented in the multiple index. "isCase" will determine whether case is important in this index. If "isCase" is TRUE, case is important. "blkSize" is the block size for the index block file. The rules for selecting the optimal block size for multiple indexes are the same as those for a field oriented index file, except that you have to accumulate the total size of all the fields represented by the multiple index when you calculate the block size. Again, it is advised that the block size be based on testing if performance is an issue.

The current index for the table is unchanged after a call to createMultiIndex().

If createMultiIndex() is successful, _OK_ is returned. If it is not, isTableError(tbl) will be true for this table and dbtblerror() will contain a description of the error.

4.3 openDBIndexes

```
int openDBIndexes (dbTable * tbl)
```

4.4 openMultiIndexes

```
int openMultiIndexes (dbTable * tbl)
```

openMultiIndexes() opens the multiple field indexes associated with the table "tbl". As the indexes are opened, the index descriptors are placed in tbl->midxList (a linked list). If openMultiIndexes() is successful it returns _OK_. If an error occurs, isTableError(tbl) will be TRUE for this table and dbtblerror(tbl) will return a description of the error. The current index for the table is unchanged by a call to openMultiIndexes().

4.5 closeDBIndexes

```
int closeDBIndexes (dbTable * tbl)
```

closeDBIndexes() closes the open indexes associated with the table "tbl". It is not an error if there are no open indexes for tbl; _OK_ is simply returned in that instance. If closeDBIndexes() is successful, all the indexes for a table will be flushed to disk and closed, and _OK_ will be returned. If an error occurs, isTableError(tbl) is TRUE for this table and dbtblerror(tbl) will return a description of the error.

4.6 addDBIndexes

```
int addDBIndexes (dbTable * tbl)
```

addDBIndexes() will add the items in tbl->fields[] to the open indexes for each field being indexed. It is not an error to call addDBIndexes() for a table that has no indexed fields; _OK_ is simply returned in that instance. A check is made before any data is added to insure that no unique constraints are violated. This way, the user can insure that no index data is altered when a call to addRecord() fails due to a unique index violation. addDBIndexes() returns _OK_ on success. If an error occurs, isTableError(tbl) will be TRUE for this table and dbtblerror(tbl) will return a description of the error.

4.7 addMultiIndexes

```
int addMultiIndexes (dbTable * tbl)
```

addMultiIndex() adds index information to the multiple field indexes associated with table "tbl". If it is successful, addMultiIndex() returns _OK_. Otherwise, isTableError(tbl) is true for this table and dbtblerror(tbl) will return a description of the error.

4.8 setCurrentIndex

```
int setCurrentIndex (dbTable * tbl, char *idxName)
```

setCurrentIndex() sets the current index pointer for the table "tbl" to the first index with the name that matches "idxName" up to strlen(idxName). If setCurrentIndex() returns _OK_, tbl->current will be set to the current dbIndex in idxList. If an error occurs, tbl->current will be unchanged, isTableError(tbl) will be TRUE for this table and dbtblerror(tbl) will return a description of the error.

5 DCDB Utility functions

This is the C file with the miscellaneous utility functions and globals needed by the cdb module.

The following is the definition of dbErrMsg[]. This is an array that contains pointers to the strings that provide descriptions for errors that can occur.

```
char *dbErrMsg[] = { "no cdb error", /* DB_NOERROR */
    "I/O permission denied",              /* I/O error - */
                                          /* access denied */
    "I/O too many open files",            /* I/O error - */
                                          /* too many files opened */
    "I/O file or directory doesn't exist",/* I/O error - */
                                          /* no such file or directory */
    "I/O bad file descriptor",            /* I/O error - */
                                          /* bad file descriptor */
    "I/O incomplete read or write",       /* DB_IO_READWRITE */
    "I/O operation prohibited",           /* DB_IO_PROHIB */
    "I/O operation would cause a deadlock",/* DB_IO_DEADLK */
    "I/O locking failed", "memory error", /* Memory error */
    "invalid cdb file",                   /* DB_INVALIDFILE */
    "list related error",                 /* DB_LISTERROR */
    "field level error",                  /* DB_FIELD */
    "index related error",                /* DB_INDEX */
    "couldn't create the index",          /* DB_CREATEINDEX */
    "cdb memory variables corrupted",     /* DB_UNSTABLE */
```

```
    "unique index constraint violated",   /* DB_UNIQUE */
    "too many indexes for this table",    /* DB_TOOMANYIDX */
    "bad function parameter",             /* DB_PARAM */
    "shell related error",                /* DB_SHELL */
    "table locked or in use",             /* DB_LOCK */
    "parser related error",               /* DB_PARSE */
    "syntax error",                       /* DB_SYNTAX */
    "bad input",                          /* DB_INPUT */
    /* Add new ones here */
    "unspecified cdb error"               /* DB_UNSPECIFIED */
};
```

5.1 getTimeStamp

```
const char *getTimeStamp (void)
```

getTimeStamp() calls asctime() to get a string representing the date and time in a consistent way. This is an ANSI function and available across platforms. The string returned by this function is saved in the table header as is. See the compiler documentation on the asctime() function for more information.

5.2 sortedTimeStamp

```
const char *sortedTimeStamp (void)
```

sortedTimeStamp() calls asctime() to get a string representing the data and time. Then, it pulls the information out and places it in the following format:

```
    YYYYMMDDHHMMSS
```

where "YYYY" is year, "MM" is month, "DD" is day, "HH" is hour, "MM" (the second set) is minutes and "SS" is seconds.

5.3 field2Record

```
void field2Record (dbTable * tbl)
```

field2Record() translates the strings in the tbl->fields[] array to the format expected in each record. The data gets copied to the tbl->data field of the tbl structure.

5.4 record2Field

```
void record2Field (dbTable * tbl)
```

record2Field() translates the data in the tbl->data field to strings. Each string is copied into the tbl->fields[] array.

5.5 dbtblerror

```
const char *dbtblerror (dbTable * tbl)
```

dbtblerror() returns a constant string that describes the error that has occurred. dbtblerror() is guaranteed to return a valid string even if an error did not occur, so it is safe to use it in a function call. The return value is a static string that is local to the dbtblerror() function. dbtblerror() should be used in a context where an error occurred at the table level (with a valid table descriptor). However, if the error is global, dbtblerror() will handle it correctly.

dbtblerror() clears all error conditions after it is called. Therefore, an error should be processed immediately after dbtblerror() is called (at least, before it or dberror() is called again) or error information could be lost.

5.6 dberror

```
const char *dberror (void)
```

dberror() returns a constant string that describes the error that has occurred. dberror() is guaranteed to return a valid string even if an error did not occur, so it is safe to use it in a function call. The return value is a static string that is local to the dberror() function. dberror() should be used in a context where the error is global (where there is no valid table descriptor). It will not handle it properly if the error is really at the table level as opposed to global.

dberror() clears all error conditions after it is called. Therefore, an error should be processed immediately after dberror() is called (at least, before it or dbtblerror() is called again) or error information could be lost.

6 DCDB Table Traversal Functions

This file contains the record traversing functions for the DCDB module. All the traversal functions set the current record pointers for the table only; no data is read into the record buffer until a call to retrieveRecord() is made. This is done for efficiency; the user does not always want to read data into the data buffers when moving through the table. However, it does place the burden of insuring that retrieveRecord() is called after each movement where the user wants to work with the data in the record.

6.1 gotoRecord

```
off_t gotoRecord (dbTable * tbl, off_t recno)
```

gotoRecord() sets the file pointer to the beginning of the record number given by “recno” in table “tbl”. If recno is too small or big, the current record number becomes 0 or number of records, respectively. No data is read from the file by a call to gotoRecord(); the file pointer is just moved to point to the record number

specified. This behavior is consistent in all the cdb move functions; no data is retrieved until retrieveRecord() is called.

On success, gotoRecord() returns the offset into the table where the current pointer is located. If an error occurs, isTableError(tbl) is TRUE for this table and dbtblerror(tbl) returns a string that describes the error.

6.2 nextRecord

```
off_t nextRecord (dbTable * tbl)
```

nextRecord() moves the file pointer to the next record in the table given by "tbl". Nothing happens if the table has no records. If the table pointer is at the end of the table, nextRecord() returns the offset to the last record and sets tbl->eof to TRUE.

nextRecord() returns the offset into the table of the current record from the beginning of the file on success. If an error occurs, isTableError(tbl) is TRUE for tbl and dbtblerror(tbl) will return a string describing the error.

6.3 nextIndexRecord

```
off_t nextIndexRecord (dbTable * tbl)
```

nextIndexRecord() takes the current record pointer for the table "tbl" to the next record as given by the index tbl->current. if tbl->current is not set, nextIndexRecord() decays to a call to nextRecord(tbl). If nextIndexRecord() results in the record pointer going off of the end of the index, the last record by the index is returned and tbl->eof is set to TRUE. If an error occurs in a call to nextIndexRecord(), isTableError(tbl) is TRUE for this table and dbtblerror() returns a string that describes the error.

6.4 prevRecord

```
off_t prevRecord (dbTable * tbl)
```

prevRecord() is the same as nextRecord() except it moves back one record in the table. If the table pointer backs up to before the first record, the offset of the first record is returned and tbl->bof is set to TRUE. On error, isTableError(tbl) is TRUE for tbl and dbtblerror(tbl) returns a string that describes the error.

6.5 prevIndexRecord

```
off_t prevIndexRecord (dbTable * tbl)
```

prevIndexRecord() works the same as nextIndexRecord() except the offset of the previous record by the current index is returned. If the current record pointer moves past the first record, the offset of the first record is returned and tbl->bof is set to TRUE. On error, isTableError(tbl) is TRUE for tbl and dbtblerror(tbl) returns a string that describes the error.

6.6 headRecord

```
off_t headRecord (dbTable * tbl)
```

headRecord() moves the file pointer to the first record in the table. No data is read from the file and the same diagnostics are returned as the other movement functions. tbl->bof and tbl->eof are set to FALSE if headRecord() is successful. On error, isTableError(tbl) is TRUE for tbl and dbtblerror(tbl) returns a string that describes the error.

6.7 headIndexRecord

```
off_t headIndexRecord (dbTable * tbl)
```

headIndexRecord() moves the record pointers to the head of the table by the current index. tbl->eof and tbl->bof are set to FALSE and an offset of the first record (by the index) is returned. If there is no current index, headIndexRecord() decays to a call to headRecord(). On error, isTableError(tbl) is TRUE for tbl and dbtblerror(tbl) returns a string that describes the error.

6.8 tailRecord

```
off_t tailRecord (dbTable * tbl)
```

tailRecord() is the same as headRecord() except it moves the file pointer to the last record in the file. tbl->eof and tbl->bof are both set to FALSE on a successful call to tailRecord(). On error, isTableError(tbl) is TRUE for tbl and dbtblerror(tbl) returns a string that describes the error.

6.9 tailIndexRecord

```
off_t tailIndexRecord (dbTable * tbl)
```

tailIndexRecord() works the same as headIndexRecord(), except the current record pointer is set to the last record by the current index. On error, isTableError(tbl) is TRUE for tbl and dbtblerror(tbl) returns a string that describes the error.

6.10 searchIndexRecord

```
off_t searchIndexRecord (dbTable * tbl, char *fieldValue)
```

searchIndexRecord() searches for the item given by "fieldValue" in the current index for the table given by "tbl". If there is no current index or the table has no records, _NOTFOUND_ is returned. On success, the tbl->eof and tbl->bof are set to FALSE and offset of the record "found" is returned. The currency pointers for the table are set to the "found" record. searchIndexRecord() uses a best fit algorithm. If an exact match is not found, searchIndexRecord() returns

the next best fit for a match. For example, if the item being searched for is less than or equal to anything else in the index, the first record by the current index is returned. Otherwise, the returned item indicates where the searched for item would be inserted on an add.

On error, isTableError(tbl) is TRUE for tbl and dbtblerror(tbl) returns a string that describes the error.

6.11 searchExactIndexRecord

```
off_t searchExactIndexRecord (dbTable * tbl, char *fieldValue)
```

searchExactIndexRecord() returns _NOTFOUND_ unless an exact match is made in the search. Otherwise, it works exactly like searchIndexRecord(). On error, isTableError(tbl) is TRUE for tbl and dbtblerror(tbl) returns a string that describes the error. See page 14 for information on properly handling errors. My recommendation is that you do something as follows:

```
setCurrentIndex (tbl, "lastname");
if (isTableError(tbl) {...handle the error...}
status = searchExactIndexRecord (tbl, "May");
if (isTableError(tbl) {...handle the error...}
if (status == _NOTFOUND_) {...didn't find it...}
else {...found it...}
```

7 DCDB Add Functions

This file contains the add functions for the cdb module.

7.1 addRecord

```
int addRecord (dbTable * tbl)
```

addRecord() adds the data in tbl->fields[] to the end of the table given by tbl->fd. The data is copied from the fields[] array to tbl->data using field2Record().

All indexes are updated before the record is added. If a unique violation would result from adding the current record, no data is added to the indexes or tables and an _ERROR_ is flagged.

Because of the checking that is done by addRecord() for unique constraint violations, it is faster to add records for a table in which there are no unique constraints.

On successful completion, addRecord() returns _OK_. If an error occurs, isTableError(tbl) will be TRUE for this table and dbtblerror(tbl) will return a string that identifies the problem.

7.2 massAddRecords

```
ListHeader *massAddRecords (dbTable * tbl, ListHeader * lh)
```

massAddRecords() adds the records stored in the list "lh" to the table "tbl". Each item in "lh" should be tbl->hdr->sizeRecord in size and should be allocated dynamically on the heap. The easiest way to populate these items is to:

1. use the setXXXXXField() functions to populate the field items for this record.
2. use field2Record() to translate from the field values to the tbl->data member.
3. allocate a block of memory of tbl->hdr->sizeRecord size.
4. use memmov() to copy tbl->data to the allocated block.
5. add the allocated block to the list.

massAddRecords() will attempt to add all the records to the table. If it is unsuccessful because of a unique constraint violation, the records that could not be added will be placed in a list and returned to the caller. It is the caller's responsibility to delete the list that is returned and to clean up the items in "lh". On successful return, all the items in "lh" were successfully added to the table, the items in the return list were not added because of unique constraints. If an error occurs, isTableError(tbl) will be TRUE for this table and dbtblerror(tbl) will return a string describing the error.

8 DCDB Edit Functions

This file contains the edit functions for the cdb module.

8.1 retrieveRecord

```
off_t retrieveRecord (dbTable * tbl)
```

retrieveRecord() retrieves the current record from the table. The current record is given by record number tbl->crec and is tbl->offset bytes from the beginning of the data file. This is the function that actually reads data from the table. The movement functions *do not* read data. So, if the user calls gotoRecord(), no data is retrieved into the tbl->data item until retrieveRecord() is called.

retrieveRecord() returns _ERROR_ on error, or the number of bytes read on success. It is an error if the number of bytes read is not the same as the record size, in which case _ERROR_ is returned. If 0 is returned, that means that the record is marked deleted and must be undeleted before the data can be retrieved. In that case, tbl->data is set to all 0's.

8.2 updateRecord

```
int updateRecord (dbTable * tbl)
```

updateRecord() updates the current record in the table "tbl" with the data stored in the tbl->fields[] strings. This is copied into the tbl->data field and written. Update first stores the original record as a backup to insure no data loss. Then, it deletes all indexes associated with the original record and replaces them with values from the new record. If that succeeds, it updates the record in the table and returns _OK_. If the replacement of indexes fails, updateRecord() restores the table data to the old record and returns _ERROR_. The calling program should determine whether an error is the result of unique constraints. If it is, the caller should be careful to reindex the table to get it back to the point of stability that it was before the call to updateRecord(). *See the sample programs for examples on how to do this.*

8.3 recordDeleted

```
int recordDeleted (dbTable * tbl)
```

recordDeleted() examines the data in tbl->data with the assumption that this data has been recently received into memory from the table. If the first byte of the record is a "*", the record is marked deleted and TRUE is returned. Otherwise, FALSE is returned.

8.4 isRecordDeleted

```
int isRecordDeleted (dbTable * tbl)
```

isRecordDeleted() retrieves the current record into tbl->data and determines whether it has been deleted. If so, it returns TRUE; FALSE is returned otherwise.

8.5 deleteRecord

```
int deleteRecord (dbTable * tbl)
```

deleteRecord() marks the current record for deletion. It first retrieves the contents of the record from the table, marks it deleted and saves it. Once marked as deleted, retrieveRecord() will not retrieve the contents of the record unless it is marked undeleted, even though the actual contents of the record remain unchanged.

deleteRecord() returns _OK_ on success, _ERROR_ on error.

8.6 undeleteRecord

```
int undeleteRecord (dbTable * tbl)
```

undeleteRecord() unmarks a record that was previously marked as deleted and then stores the record to disk. After it is unmarked, the data can be retrieved successfully by retrieveRecord(). This function returns _OK_ on success, _ERROR_ on error.

9 DCDB Pack and Reindex Routines

These are the routines that allow the user to pack a DCDB table and reindex a DCDB table.

9.1 packTable

```
dbTable *packTable (dbTable * tbl, int save)
```

The packTable() function packs the records in the table "tbl". Any records flagged as deleted are not copied to the new table when it is packed. If "save" is FALSE, the old table is deleted and replaced by the new, packed table. If "save" is TRUE, a copy of the old table without indexes will be saved in the file ".cdbpack.packTable.tmp.db". This can then be stored by the application in whatever form is required. The file name ".cdbpack.packTable.tmp.db" is used for each pack in which the file is saved, so if something is not done with the backup copy of the table right away, it will be deleted and replaced.

Upon success, packTable() returns a pointer to an open, newly packed table. If an error occurs, isDBError() will be TRUE and dberror() will return a string that describes the error.

It should be noted that packing a table can take considerable time, depending on how many records are in the table. Also, the system tends to become bogged down reading from and writing to the same disk (disk contention). Therefore, packing of tables should be done sparingly for performance sake.

9.2 reindexTable

```
int reindexTable (const char *tblname)
```

reindexTable() reindexes the closed table given by "tblname". The table is opened without index support, all indexes associated with the table are deleted and then are rebuilt. Upon successful completion, the table and associated (newly built) indexes are closed and _OK_ is returned. If an error occurs, isDBError() is TRUE and dberror() will return a string that describes the error.

Reindexing is a time-consuming process that should be used sparingly for performance sake. Also, it is possible that an error in the process of reindexing a table could leave the table without indexes. However, this is no worse than having the data table with corrupted indexes (which is, conceivably, why you would use reindexTable()).

10 DCDB User Interface Functions

These are the user interface functions of the DCDB library, functions that are designed purely as candy for the end-user. These functions are not necessary to using DCDB and the functionality provided here can be had through other means.

10.1 buildTable

```
dbTable *buildTable (const char *tblname, const char *tblinfo,
             fieldType * ftype, char **names,
             int *flens, int *declens)
```

buildTable() performs an identical function to createTable(). It does not, however, require a person to create on the heap and initialize a dbHeader record and a dbField array.

"tblname" is the name of the table to create. "tblinfo" is a string of data that will be saved in the table of max length TABLE_INFO_WIDTH. This allows the user to store in the table information that is specific to the application. "ftype" is an array of field types, each describing what type of field is represented in the table. "names" is an array of character strings that contains names for each field. "flens" is an array of integers that gives the length for each field. And, "declens" is the length of the decimal part of each field, if that is applicable (only applies to NUMBER fields). The "ftype", "names", "flens" and "declens" arrays contain the information that fully identifies each field. These must have the same number of items and must correspond with each other to describe a field. The "names" array *must* be NULL terminated. If it isn't, you will probably get a protection fault on a call to buildTable().

Upon success, buildTable() will return the descriptor for the open table. If an error occurs, isDBError() will be TRUE and dberror() will return a string which describes the error. If that happens, buildTable() will return NULL.

It is recommended that upon successful completion of the buildTable() function that the user then create all indexes for the table, close the table (with a call to closeTable()) and then reopen the table with a call to openTable(). This should be done before any data is added to the table. Closing the table and reopening it is a defensive technique that will insure that all the basic file structures are flushed to disk and reinitialized at reopening.

Note: this function is deprecated, as is createTable(), by the parseDBDef() function. Tables, indexes and workspaces should be created by creating a .df file (either in the application or manually) and then calling parseDBDef() with the name of the .df file as an argument.

10.2 getFieldNum

```
int getFieldNum (dbTable * tbl, const char *fieldName)
```

getFieldNum() is a simple function that is used to convert a field name to a numerical offset into the tbl->fields array. "tbl" is the table to work with and "fieldName" is the name of the field. If the call is successful, getFieldNum() returns an integer value "i" such that tbl->fields[i] is the storage space for the field, tbl->flens[i] is the length and tbl->declens[i] is the length of the decimal part, if if applicable. It should be noted that getFieldNum() uses strncmp() to compare "fieldName" with the names of the fields, so the field doesn't have to match exactly. Therefore, if you have a field named "line" and one named "lineNum" and "fieldName" is "line", you may not get what you expect from getFieldNum(). It is better to make your fields distinct within the first few characters to avoid such ambiguities. At the least, you should have the "line" field in the table before the "lineNum" field, otherwise you will never get a valid return from getFieldNum() for the "line" field.

If an error occurs, isTableError(tbl) is TRUE for this table and dbtblerror(tbl) returns a string that describes the error.

10.3 setCharField

```
int setCharField (dbTable * tbl, const char *fieldName, const char *value)
```

setCharField() sets the field given by "fieldName" in the table "tbl" to "value". It is an error if the field is not a character field or if there is no field called "fieldName". setCharField() insures that the data copied to the field buffer is of tbl->flens[] length for that field, at most. If "value" is NULL, the characters in the field are set to all binary 0 values.

If an error occurs, isTableError(tbl) is true for this table and dbtblerror(tbl) returns a string describing the error.

10.4 setNumberField

```
int setNumberField (dbTable * tbl, const char *fieldName, double value)
```

setNumberField() sets the NUMBER field given by "fieldName" in the table "tbl" to the value "value". It is an error if the field with the name "fieldName" is not a numeric field or if there is no field with that name. The data is copied to the field right justified and padded with spaces on the left. If "value" is 0, the field is set to "0", right justified and padded with spaces.

If an error occurs, isTableError(tbl) is true for this table and dbtblerror(tbl) returns a string describing the error.

10.5 setIntField

```
int setIntField (dbTable * tbl, const char *fieldName, int value)
```

setIntField() sets the NUMBER field given by "fieldName" in the table "tbl" to the value "value". It is an error if the field with the name "fieldName" is not a numeric field or if there is no field with that name. The data is copied to the field right justified and padded with spaces on the left. If "value" is 0, the field is set to "0", right justified and padded with spaces.

If an error occurs, isTableError(tbl) is true for this table and dbtblerror(tbl) returns a string describing the error.

10.6 setDateField

```
int setDateField (dbTable * tbl, const char *fieldName, const char *value)
```

setDateField() sets the DATE field given by "fieldName" in the table "tbl" to the value "value". It is an error if the field with the name "fieldName" is not a date field, if there is no field with that name or if the date field is not 8 digits long. If "value" is NULL, the date field is given today's date.

If an error occurs, isTableError(tbl) is true for this table and dbtblerror(tbl) returns a string describing the error.

10.7 setDateField2TimeStamp

```
int setDateField2TimeStamp (dbTable * tbl, const char *fieldName,
                const char *timeStamp)
```

setDateField2TimeStamp() sets the DATE field given by "fieldName" in the table "tbl" to the value in "timeStamp". setDateField2TimeStamp() converts the time stamp to an 8 digit date in the form YYYYMMDD before it sets the field. It is an error if the field with the name "timeStamp" is not a valid time stamp or if there is no field with that name.

If an error occurs, isTableError(tbl) is true for this table and dbtblerror(tbl) returns a string describing the error.

10.8 setLogicalField

```
int setLogicalField (dbTable * tbl, const char *fieldName, const char value)
```

setLogicalField() sets the LOGICAL field given by "fieldName" in the table "tbl" to the value "value". It is an error if the field with the name "fieldName" is not a numeric field, if there is no field with that name, or if value is not one of 'y', 'Y', 't', 'T', 'n', 'N', 'f', 'F'. If "value" is 0, the field is set to 'T'.

If an error occurs, isTableError(tbl) is true for this table and dbtblerror(tbl) returns a string describing the error.

10.9 setTimeStampField

```
int setTimeStampField (dbTable * tbl, const char *fieldName,
               const char *timeStamp)
```

setTimeStampField() sets the TIMESTAMP field given by "fieldName" in the table "tbl" to the value "timeStamp". A time stamp is a string like that returned from asctime(), as follows: "Tue Mar 21 11:10:42 MST 2000" This gives the date/time information for a particular moment in time. It is an error if the field with the name "timeStamp" is not a timeStamp field, if there is no field with that name or if the timeStamp field is not 25 digits long. If "timeStamp" is NULL, the timestamp is set to now.

If an error occurs, isTableError(tbl) is true for this table and dbtblerror(tbl) returns a string describing the error.

10.10 setDefaultField

```
int setDefaultField (dbTable * tbl, const char *fieldName)
```

setDefaultField() sets the field given by "fieldName" in the table "tbl" to a default value. What a "default" value is depends on the field. This is described in setCharField(), setNumberField(), setLogicalField(), setDateField() and setTimeStampField() for calls with the "value" or "timeStamp" parameter set to NULL.

If an error occurs, isTableError(tbl) is true for this table and dbtblerror(tbl) returns a string describing the error.

10.11 getField

```
char *getField (dbTable * tbl, const char *fieldName)
```

getField() gets the value that is currently set in the field given by "fieldName" in the table given by "tbl". The value is returned in a pointer to the field. This should be considered a constant value to the calling program. It is an error if "fieldName" is not the name of a valid field in "tbl".

If an error occurs, isTableError(tbl) is true for this table and dbtblerror(tbl) returns a string describing the error.

11 DCDB Definition File Functions

A definition file is a text file (similar to a script) that tells the definition processor what to do. The allowable actions are:

- *create table* Creates a table and the associated indexes. The generic syntax is as follows:

```
create table "tablename.db"
  info "Info string - application dependent"
{
   "charfield" char (num);
   "numfield" number (num:dec);
   "logicalfield" logical;
   "datefield" date;
   "timestamp" time;
} indexed {
   idx "idxname" blksz:[case]nocase:[dup]unique "fname";
   midx "midxname" blksz:[case]nocase "fld1","fld2",...;
};
```

Note: if there weren't any indexes for this table, the user would simply replace the 'indexed' keyword with ';' and that would be the end of the table definition.

- *create workspace* Creates a workspace. The syntax for this command is as follows:

```
create workspace "workspacename.ws";
```

- *add* Adds a table to a workspace. The syntax for this command is as follows:

```
add "tablename.db" to "workspace.ws" index "currentidx";
```

This adds "tablename.db" to the workspace "workspace.ws" with the index "currentidx" the current index.

- comments Comments are allowed anywhere in the definition file. Both C and C++ comments are recognized and supported.

The test program "flogws2.c" works identically to "flogws.c" except it uses a definition file to create the tables, indexes and workspace and bind them together. Then, it opens the workspace file, gets table information from it and proceeds to work. Following is the definition file used to create the objects:

```
//  Definition File:    flogws2.df
//
//  Creates tables, indexes and the
//  workspace for flogws2 test routine.

create table "users.db"
  info "Primary:Logistical information for users"
```

```
{
    "ssnumber" char (9);
    "dob" date;
    "age" number (3:0);
    "street" char (65);
    "street2" char (65);
    "city" char (50);
    "state" char (2);
    "zip" char (10);
} indexed {
    idx "userssnidx" 256:case:unique "ssnumber";
};

create table "comments.db"
  info "Secondary:Text information for comments on users"
{
    "ssnumber" char (9);
    "lineNum" number (4:0);
    "cline" char (80);
} indexed {
    midx "commentsidx" 256:case "ssnumber", "lineNum";
};

create workspace "flogws2.ws";

add "users.db" to "flogws2.ws" index "userssnidx";
add "comments.db" to "flogws2.ws" index "commentsidx";
```

When the parser has gone through the first "create table" expression, the table "users.db" and the index file "userssnidx.idx" would have been created. After the second "create table" expression, "comments.db" and "commentsidx.idx" will have been created. Then, the workspace file "flogws2.ws" is created by the "create workspace" command. Finally, "users.db" is added to "flogws2.ws" with the index "userssnidx" the current index and "comments.db" is added to "flogws2.ws" with "commentsidx" as the current index.

After this definition file is parsed and executed, the programmer need only open "flogws2.ws" to get access to the tables and indexes that are added to the workspace.

11.1 parseDBDef

```
int parseDBDef (const char *file)
```

parseDBDef() parses the definition file given by "file". If it succeeds it returns _OK_ and the objects specified in the definition file will have been created. If it fails, it returns _ERROR_. Because parseDBDef() performs the actions as it proceeds (as an interpreter would), it is possible for parseDBDef() to have

created some of the objects in the definition file at the point when a syntax error occurs. Any clean up that is required is the responsibility of the caller.

12 DCDB Workspace Functions

A workspace is basically a grouping of tables. The workspace functionality is provided as a convenience. The user can create the tables necessary for an application, create the supporting indexes and then add the tables to a workspace. Once that is done, the user only needs to open and close the workspace to open and close all the tables and indexes that are part of the workspace.

12.1 wsCreate

```
workSpace *wsCreate (const char *name)
```

wsCreate() creates a workspace with name "name". wsCreate() simply sets up the internal variables for a workspace and saves them to a file. If it is successful, wsCreate() returns a valid workSpace descriptor that can be used in further workspace function calls.

If an error occurs, isDBError() is TRUE and dberror() returns a string that describes the error.

12.2 wsAddTable

```
int wsAddTable (workSpace * ws, dbTable * tbl, char *comment)
```

wsAddTable() adds the table "tbl" to the workspace "ws". The character string "comment" allows the application designer to store information that is application dependent with the table information in the workspace. If wsAddTable() is successful, the current index (tbl->current) becomes the default index and is set as current each time the workspace opens the table. wsAddTable() returns _OK_ if it succeeds.

If an error occurs, isDBError() is TRUE and dberror() returns a string that describes the error.

12.3 wsClose

```
int wsClose (workSpace * ws)
```

wsClose() closes the workspace given by the descriptor "ws". All the tables and indexes grouped with "ws" are closed as well. wsClose() returns _OK_ if it is successful.

If an error occurs, isDBError() is TRUE and dberror() returns a string that describes the error.

12.4 wsOpen

```
workSpace *wsOpen (const char *name)
```

wsOpen() opens the workspace with the file name "name". All tables and indexes associated with the workspace are opened and default indexes are set for each table. If wsOpen() succeeds, it returns a valid workspace descriptor; if an error occurs, the descriptor is not valid (it is NULL), isDBError() is TRUE and dberror() returns a string describing the error.

12.5 wsGetTable

```
dbTable *wsGetTable (workSpace * ws, char *name)
```

wsGetTable() returns a descriptor to the table with the file name "name" from the workspace "ws". It is not an error if there is no table with that name in the workspace; NULL is simply returned in that case. If an error occurs, isDBError() is TRUE and dberror() returns a string describing the error.

13 High-level C API Functions

This section discusses the high-level interface functions for C programmers. This is the recommended method for working with DCDB using C programs. There are numerous examples of using DCDB with this Application Programmer Interface (API) in (topdir)/tests/c. You should study this C code to see how things are done using the high-level APIs. What follows should just be considered reference information.

Error conditions are communicated via function return values. If an error occurs, as indicated by the return value, the variable cdberror will have a description of the error. cdberror is defined as follows:

```
char cdberror[RESULT_SIZE + 1];
```

13.1 dbcreate

```
int dbcreate (char *dffile)
```

The following is an example that shows how you would use dbcreate() to create tables and indexes:

```
int status;
status = dbcreate("deffile.df");
if (status == -1) {
  Process the error...
```

```
}
```

dbcreate creates the tables and workspaces defined in the definition file given by "deffile.df". If any error occurs, an error (-1) is returned. If dbcreate is successful, 0 is returned.

For practical examples of using dbcreate, you should study the code in (topdir)/tests/c. Basically, the scripts do the following:

- open a file (flogadd1.df, for example),
- write the code into the file to create the needed tables and indexes,
- close the file,
- call dbcreate with the file name as an argument ("dbcreate flogadd1.df", for example),
- and remove the .df file

Embedding this logic into your program is the simplest way to insure that the .df is created when needed.

13.2 dbopen

```
const char *dbopen (char *tbl)
```

dbopen opens the table given by "tbl". It is an error if the table doesn't exist. Upon success, dbopen returns a read-only pointer to the name of a handle. That handle is to be used for all other table access functions. This value must be copied to a char array, because many of the access functions use the same static storage space to return results as the array the handle name is returned in. The handle is really just the name of the table, so in a pinch you can replace the handle name with "tbl" (in this example).

If an error occurs, dbopen returns a NULL pointer and cdberror contains a description of the error.

13.3 dbclose

```
int dbclose (char *tbl)
```

dbclose closes the table given by the argument "tbl", which is actually a table handle. It is an error if the table is not open. Upon successful completion, a table and all it's associated indexes will be properly flushed to disk and closed and 0 is returned. If an error occurs, dbclose returns -1 and cdberror will contain a read-only string that describes the error.

It is entirely permissible to use dbexit to close all open tables right before exiting, should you choose to do so.

13.4 dbnumrecs

```
int dbnumrecs (char *tbl)
```

dbnumrecs returns the number of records currently being managed in the table. It is an error if the table is not open yet. Upon successful completion, dbnumrecs restores the number of records. If an error occurs, an exception is raised and a string is returned that describes the error.

13.5 dbnumfields

```
int dbnumfields (char *tbl)
```

dbnumfields returns the number of fields in the table given by "tbl" (this is a table handle). It is an error if the table is not open yet. Upon successful completion, dbnumfields returns the number of fields. If an error occurs, dbnumfields returns -1 and cdberror contains a read-only string that describes the error.

13.6 dbseq

```
int dbseq (char *tbl)
```

dbseq returns the "nextSequence" value stored in the table header of the table given by "tbl" (a table handle). It then increments the "nextSequence" value. It is an error if the table is not open. Upon successful completion, dbseq returns the sequence value stored in the table before the call was made. On error, a -1 is returned and cdberror contains a read-only string that describes the error.

13.7 dbiseof

```
int dbiseof (char *tbl)
```

dbiseof returns 1 if the end of file indicator is TRUE for the table given by "tbl" (a table handle) and 0 if it is FALSE. It is an error if the table is not open. On error, -1 is returned and cdberror contains a read-only string that describes the error.

13.8 dbisbof

```
int dbisbof (char *tbl)
```

dbisof returns 1 if the beginning of file indicator is TRUE for the table given by "tbl" (a table handle) and 0 if it is FALSE. It is an error if the table is not open. On error, -1 is returned and cdberror contains a read-only string that describes the error.

13.9 dbshow

```
char *dbshow (const char *tbl, const char *fldnm)
```

dbshow is used to get the value of the field in the table "tbl" (a table handle). The field is given by "fldnm". If successful, dbshow returns the information in a read-only static char string that is overwritten after each call to dbshow. If there is an error, dbshow returns a NULL pointer and cdberror contains a string that describes the error.

13.10 dbflen

```
int dbflen (char *tbl, char *fldnm)
```

dbflen returns the length of the field "fldnm" in the table "tbl" (a table handle). If an error occurs, dbflen returns -1 and cdberror is a read-only string that describes the error.

13.11 dbdeclen

```
int dbdeclen (char *tbl, char *fldnm)
```

dbdeclen returns the length of the decimal portion for the field given by "fldnm". If the field is not numeric, this will always be 0. It is an error if the table is not open or "fldnm" is not a valid field name. If an error occurs, an error is raised and a string describing the error is returned. The argument "tbl" is a table handle.

13.12 dbsetint

```
int dbsetint (char *tbl, char *fldnm, int val)
```

dbsetint sets the tbl->fields[] value for the field given by "fldnm" in the table given by "tbl" (this is a table handle) to "val" in this example. The table must be open and "fldnm" must be a valid numerical field. Upon successful completion, the field is set and 0 is returned. If an error occurs, -1 is returned and cdberror contains a read-only string describing the error.

13.13 dbsetnum

```
int dbsetnum (char *tbl, char *fldnm, double val)
```

dbsetnum works as dbsetint except it accepts a double value field. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

13.14 dbsetlog

```
int dbsetlog (char *tbl, char *fldnm, char val)
```

dbsetlog is used to set a logical field in the tbl->fields[] array of the table "tbl" (a table handle). Other than that, it functions identically to dbsetint. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

13.15 dbsetchar

```
int dbsetchar (char *tbl, char *fldnm, char *val)
```

dbsetchar works as dbsetint except it accepts a string value field. If an error occurs, -1 is returned and cdberror is a read-only string describing the error. If the "val" variable is set to "0", The value of "fldnm" is set to "" (empty string).

13.16 dbsetdate

```
int dbsetdate (char *tbl, char *fldnm, char *val)
```

dbsetdate works as dbsetint except it accepts a date value field. If an error occurs, -1 is returned and cdberror is a read-only string describing the error. If the "val" argument is "0", the date value is set to today's date.

13.17 dbsettime

```
int dbsettime (char *tbl, char *fldnm, char *val)
```

dbsettime works as dbsetint except it accepts a time stamp value. If an error occurs, -1 is returned and cdberror is a read-only string describing the error. If "val" is "0", the current date and time is used to generate the timestamp.

13.18 dbadd

```
int dbadd (char *tbl)
```

dbadd adds the data placed in the tbl->fields[] array using the dbsetxxx function calls. It is an error if a unique index constraint would be violated by the add; however, it is application dependent whether this should result in the program exiting. It is an error if the table is not open. Upon successful completion, the data will have been added to the table. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

13.19 dbretrieve

```
int dbretrieve (char *tbl)
```

dbretrieve retrieves the data for the current record into the tbl->fields[] array where it is accessible via calls to dbshow. It is an error if the table is not open. If the current record is marked for deletion, "0" is returned. You should be aware that because you move to a record doesn't mean the record is retrieved. You must call dbretrieve after movement functions in order to get the data from the table record. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

dbupdate replaces the contents of the current record with the values stored in the tbl->fields[] array of the table given by the tablehandle "tbl". It is an error if the table is not open. If an error occurs, an error is raised and a string describing the error is returned.

13.20 dbdel

```
int dbdel (char *tbl)
```

dbdel marks the current record in the table for deletion and returns 0 if successful. The record is not really deleted until dbpack is run on the table, at which point the deletion is irreversible. It is an error if the table is not open. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

13.21 dbundel

```
int dbundel (char *tbl)
```

dbundel unmarks the current record for deletion, if it is marked. If it is not marked, dbundel does nothing. It is an error if the table is not open. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

13.22 dbisdeleted

```
int dbisdeleted (char *tbl)
```

dbisdeleted returns "1" if the current record in the table is marked for deletion, "0" otherwise. It is an error if the table is not open. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

13.23 dbdbfldname

```
char *dbfldname (char *table, int fldnum)
```

dbfldname returns the name of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is returned. An error is likewise returned if there are no open tables, if the file handle is invalid or on some other critical error. If there is an error, a NULL pointer is returned and cdberror contains a read-only string describing the error.

13.24 dbfldtype

```
char *dbfldtype (char *table, int fldnum)
```

dbfldtype returns the type of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is returned. An error is likewise returned if there are no open tables, if the file handle is invalid or on some other critical error. If there is any error, a NULL pointer is returned and cdberror contains a read-only string describing the error.

13.25 dbfldlen

```
int dbfldlen (char *table, int fldnum)
```

dbfldlen returns the length of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is returned. An error is likewise returned if there are no open tables, if the file handle is invalid or on some other critical error. If there is an error, -1 is returned and cdberror contains a read-only string describing the error.

13.26 dbflddec

```
int dbflddec (char *table, int fldnum)
```

dbflddec returns the length of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is returned. An error is likewise returned if there are no open tables, if the file handle is invalid or on some other critical error. If there is an error, -1 is returned and cdberror contains a read-only string describing the error.

13.27 dbcurrent

```
int dbcurrent (char *tbl, char *idxnm)
```

dbcurrent sets the current index for the table to "indexname" and returns 0 on success. It is an error if the table is not open or if "indexname" is not a valid index for the table. If an error occurs, an error is returned and a string describing the error is made available in cdberror.

13.28 dbgo

```
int dbgo (char *tbl, int recno)
```

dbgo sets the current record to the record numbered "recno" (a 32-bit integer value) on success and returns the record number of the current record. It is an error if the table is not open or "recno" is less than one or greater than the number of records. If an error occurs, an error is returned and a string describing the error is returned.

13.29 dbnext

```
int dbnext (char *tbl)
```

dbnext sets the current record in the table to the next physical record without regard for indexing. It is an error if the table is not open. If the current record is already the last record when dbnext is called, dbnext sets the end of file marker on the table to TRUE (which can be checked with a call to dbiseof). If an error occurs, an error is returned and cdberror contains a string describing the error.

13.30 dbnextindex

```
int dbnextindex (char *tbl)
```

dbnextindex sets the current record in the table to the next record by the current index. It is an error if the table is not open. If there is no current index, dbnextindex decays to a call to dbnext. If the current record is already the last record when dbnextindex is called, dbnextindex sets the end of file marker on the table to TRUE (which can be checked with a call to dbiseof). If an error occurs, an error is returned and cdberror contains a string describing the error.

13.31 dbprev

```
int dbprev (char *tbl)
```

dbprev sets the current record in the table to the previous physical record. It is an error if the table is not open. If the current record is already the first record when dbprev is called, dbprev sets the beginning of file marker on the table to TRUE (which can be checked with a call to dbisbof). If an error occurs, an error is returned and cdberror contains a string that describes the error.

13.32 dbprevindex

```
int dbprevindex (char *tbl)
```

dbprevindex sets the current record in the table to the previous record by the current index. It is an error if the table is not open. If the current index is not set, dbprevindex decays to a call to dbprev. If the current record is already the first record when dbprevindex is called, dbprevindex sets the beginning of file marker on the table to TRUE (which can be checked with a call to dbisbof). If an error occurs, an error is returned and cdberror contains a string that describes the error.

13.33 dbhead

```
int dbhead (char *tbl)
```

dbhead sets the current record in the table to the first physical record. It is an error if the table is not open. dbhead clears the BOF and EOF markers for the table (sets them to false). if an error occurs, an error (-1) is returned and cdberror contains a string that describes the error.

13.34 dbheadindex

```
int dbheadindex (char *tbl)
```

dbheadindex sets the current record in the table to the first record by the current index. It is an error if the table is not open. If there is no current index set, dbheadindex decays to a call to dbhead. dbheadindex clears the BOF and EOF markers for the table (sets them to false). If an error occurs, an error is returned and cdberror contains a string that describes the error.

13.35 dbtail

```
int dbtail (char *tbl)
```

dbtail sets the current record in the table to the last physical record. It is an error if the table is not open. dbtail clears the BOF and EOF markers for the table (sets them to false). If an error occurs, an error is returned and cdberror contains a string describing the error.

13.36 dbtailindex

```
int dbtailindex (char *tbl)
```

dbtailindex sets the current record in the table to the last record by the current index. It is an error if the table is not open. If there is no current index, dbtailindex decays to a call to dbtail. dbtailindex clears the BOF and EOF markers for the table (sets them to false). If an error occurs, an error is returned and cdberror contains a string that describes the error.

13.37 dbsearchindex

```
int dbsearchindex (char *tbl, char *val)
```

dbsearchindex searches the table "tbl" using the current index for the value given by "val". dbsearchindex does a best fit search and only returns "0" if an error occurs. So, for example, if "val" is "111111111" and that value doesn't exist in the table but "111111112" does, then the current record pointer would point to that record after a call to dbsearchindex. dbsearchindex clears the end of file and beginning of file conditions. It is an error if the table is not open or it doesn't have a current index set. Also, you will get an error condition if you call dbsearchindex on an empty table. If an error occurs, an error is returned and cdberror contains a string that describes the error.

13.38 dbsearchexact

```
int dbsearchexact (char *tbl, char *val)
```

dbsearchexact searches the table "tbl" using the current index for the value given by "val", in this example. This is a bogus Social Security Number. dbsearchexact does an exact search and returns "0" if the item is not found. It is an error if the table is not open or it doesn't have a current index set. If the item is found, a 1 is returned and the current record is the record that has that value. If the item is not found or the table is empty, a 0 is returned. If there is an error, -1 is returned. If an error occurs, an error is returned and cdberror contains a string that describes the error.

13.39 dbpack

```
int dbpack (char *tbl)
```

dbpack packs the open table given by the handle "tbl" and returns 0 if successful. If an error occurs, an error is returned (-1) and cdberror contains a string describing the error.

13.40 dbreindex

```
int dbreindex (char *tbl)
```

dbreindex reindexes the table given by tablename and returns 0 if successful. It is an error if the table *is* open. So, "tbl" in the dbreindex call is not a table handle. If an error occurs, an error is returned (-1) and cdberror contains a string that describes the error.

13.41 dbexit

```
int dbexit (void)
```

dbexit is the only DCDB cint command that does not take any arguments. If there are no open tables, it simply returns 0. If there are open tables, it closes all of them, cleans up the list used to store table handles and returns 0.

13.42 dbtime

dbtime returns a double value that represents the number of seconds since the epoch in fairly high resolution (if your architecture supports it).

13.43 dbnumidx

```
int dbnummidx (char *tbl)
```

dbnummidx returns the number of multi-field indexes in the table given by tablehandle. It is not an error if there are no multi-field indexes for the table. If an error occurs, dbnummidx returns -1 and cdberror contains a string that describes the error.

13.44 dbismidx

```
int dbismidx (char *tbl)
```

dbismidx returns TRUE (1) if there are multi-field indexes associated with the table given by tablehandle table. It returns FALSE (0) otherwise.

13.45 dbmidxname

```
char *dbmidxname (char *tbl, int midxnum)
```

dbmidxname returns the name of the num'th multi-field index for the table given by tablehandle. It is an error if there are no multi-field indexes for the table. Use dbismidx first to determine if there are multi-field indexes for the table. *Note: the result of this call should be used immediately as subsequent calls to certain functions will overwrite it.*

13.46 dbmidxblksz

```
int dbmidxblksz (char *tbl, char *midxnm)
```

dbmidxblksz returns the blocksize of the multi-field index midxname ("midxnm") attached to the table given by tablehandle table ("tbl"). It is an error if the table has no multi-field indexes or if it has none called midxname. If an error occurs, -1 is returned and cdberror contains a string that describes the error.

13.47 dbmidxnumfldnames

```
int dbmidxnumfldnames (char *tbl, char *midxnm)
```

dbmidxnumfldnames returns the number of fields that are used to generate the multi-field index midxname ("midxnm") which is attached to the table given by tablehandle table ("tbl"). It is an error if the table has no multi-field indexes attached to it or if it has none called midxname. If an error occurs, -1 is returned and cdberror contains a string that describes the error.

13.48 dbmidxfldname

```
char *dbmidxfldname (char *tbl, char *midxnm, int num)
```

dbmidxfldname returns the num'th field name that makes up the multi-index midxname ("midxnm") which is attached to the table given by tablehandle "tbl". It is an error if the table has no multi-field indexes attached to it or if it has none named "midxnm" in this case. It is also an error if the multi-field index given by midxname doesn't have num fields (0 in this example). You should call dbmidxnumfldnames to determine how many field names are used in generating the multi-field index. If an error occurs, 0 (NULL) is returned and cdberror contains a string that describes the error. *Note: the result of this call should be used immediately as subsequent calls to certain functions will overwrite it.*

13.49 dbisindexed

```
int dbisindexed (char *tbl, char *fldnm)
```

dbisindexed returns TRUE (1) if the field name fieldname in the table given by the tablehandle table is indexed. It returns FALSE (0) otherwise. It is an error if fieldname ("fldnm") does not exist in the table given by "tbl". If an error occurs, -1 is returned and cdberror contains a string that describes the error.

13.50 dbidxblksz

```
int dbidxblksz (char *tbl, char *fldnm)
```

dbidxblksz returns the block size of the index generated for the field "fldnm" in the table given by the tablehandle "tbl". It is an error if the field is not indexed. You should call dbisindexed to insure that the field is indexed before you call dbidxblksz. If an error occurs, -1 is returned and cdberror contains a string that describes the error.

13.51 dbshowinfo

```
char *dbshowinfo (char *tbl)
```

dbshowinfo returns the contents of the info string that is stored in the table header of the table given by "tbl". This is specific to the application and has no meaning as far as DCDB is concerned. It can be a string of up to 125 characters and can contain any value the application programmer desires. It is set with the "info" portion of the "create" command in a .df file. *Note: the result of this call should be used immediately as subsequent calls to certain functions will overwrite it.*

13.52 dbteststring

```
char *dbteststring (int len, int rlen)
```

dbteststring returns a psuedo-random string of characters, all lower-case. It takes 2 arguments, both integers. If the first argument is set, the second must be 0. Conversely, if the second argument is set, the first must be 0. Setting the first argument gives you a string of that length. For example, "dbteststring(10,0)" will return a 10 character string. Setting the second argument will return a string of somewhat random length between 5 and the length specified. So, "dbteststring(0,10)" gives you a random string of lower-case letters between 5 and 10 digits long. It is an error if both arguments are 0 or if both are non-zero. If an error occurs, -1 is returned and cdberror contains a string that describes the error. *Note: the result of this call should be used immediately as subsequent calls to certain functions will overwrite it.*

13.53 dbtestupperstring

```
char *dbtestupperstring (int len, int rlen)
```

dbtestupperstring works the same as dbteststring except that the string returned will consist of upper-case alphabetical digits.

13.54 dbtestmixedstring

```
char *dbtestmixedstring (int len, int rlen)
```

dbtestmixedstring works the same as dbteststring except that the string returned will be of mixed upper and lower-case alphabetical digits.

13.55 dbtestnumber

```
char *dbtestnumber (int len, int rlen)
```

dbtestnumber works the same as dbteststring except that it returns a string of numerical digits.

13.56 dbtestnumstring

```
char *dbtestnumstring (int len, int rlen)
```

dbtestnumstring works the same as dbteststring except that it returns a string of combined numerical and upper-case alphabetical digits.

13.57 dbencrypt

```
char *dbencrypt (char *data, int size, char *pwd)
```

dbencrypt encrypts the data in the data argument (data) using the password in the passwd argument (pwd). The size argument (size) should be a multiple of 8, although if it is not, dbencrypt will find the closest multiple of 8 greater than size. The maximum size for a data item is 2048 bytes. The maximum length of the password is 56 bytes (448 bits).

dbencrypt uses the blowfish algorithm. This algorithm is unencumbered with patents and *restrictive* copyrights. It is an open source implementation written by Bruce Schneier (thanks, Bruce). You can use dbencrypt to create and manage passwords or to encrypt data in blocks.

13.58 dbdecrypt

```
char *dbdecrypt (char *data, int size, char *pwd)
```

dbdecrypt decrypts the data given by the data argument (data) using the password given by passwd (pwd). The data argument should contain data that was encrypted with dbencrypt and the passwd argument should have the same password used to encrypt the data. The size argument (size) should also be the size of the data before it was encrypted.

13.59 crc32sum

```
char *crc32sum (char *fname)
```

crc32sum will generate the crc32 check sum of the file given by fname and return that value. This is the 32 bit check sum code from /usr/src/linux/fs/jffs2/crc32[h,c] from the 2.4.18 kernel modified for my use here.

13.60 BC number engine

One of the weaknesses of TCL as a scripting language is the lack of a useful way to add, subtract, multiply and divide numbers in a way that is friendly for dollar amount calculations. This is necessary in order to use TCL for dollar-based calculations. The following bindings are available to help solve that problem. They use the GNU BC number engine (number.h and number.c), which was written in such a way that it was trivial to include it into this project. Please read the copyright notices on the source files and redestribute it according to those notices.

13.61 bcnumadd

```
char *bcnumadd (char *val1, char *val2, int scale)
```

bcnumadd uses the GNU bc number engine to add val1 to val2 in the scale given by scale.

13.62 bcnumsub

```
char *bcnumsub (char *val1, char *val2, int scale)
```

bcnumadd uses the GNU bc number engine to subtract val2 from val1 in the scale given by scale.

13.63 bcnumcompare

```
int bcnumcompare (char *val1, char *val2, int scale)
```

bcnumcompare uses the GNU bc number engine to compare val1 to val2 in the scale given by scale.

13.64 bcnummultiply

```
char *bcnummultiply (char *val1, char *val2, int scale)
```

bcnummultiply uses the GNU bc number engine to multiply val1 by val2 in the scale given by scale.

13.65 bcnumdivide

```
char *bcnumdivide (char *val1, char *val2, int scale)
```

bcnumdivide uses the GNU bc number engine to divide val1 to val2 in the scale given by scale.

13.66 bcnumraise

```
char *bcnumraise (char *val1, char *val2, int scale)
```

bcnumraise uses the GNU bc number engine to raise val1 to the power of val2 in the scale given by scale. val2 is truncated to an integer value before val1 is raised to that power.

13.67 bcnumiszero

```
int bcnumiszero (char *val1)
```

bcnumiszero uses the GNU bc number engine to determine if val1 is zero and returns 1 if it is and 0 if it is not.

13.68 bcnumisnearzero

```
int bcnumisnearzero (char *val1, int scale)
```

bcnumisnearzero uses the GNU bc number engine to determine if val1 is near zero (determined by scale). It returns 1 if it is and 0 if it is not.

13.69 bcnumisneg

```
int bcnumisneg (char *val1)
```

bcnumisneg uses the GNU bc number engine to determine if val1 is negative. It returns 1 if it is and 0 if it is not.

13.70 bcnumuninit

```
int bcnumuninit (void)
```

14 DCDB Bindings

DCDB was designed to be embedded into applications easily. This design facilitates using DCDB in a scripting language, like Tcl or Perl. In fact, with the original distribution of DCDB, I included hand coded Tcl/Tk bindings. However, with SWIG, it is unnecessary to do that.

SWIG is the Simplified Wrapper and Interface Generator (http://www.swig.org). It provides a simplified interface for creating bindings between various scripting languages and C or C++ libraries. SWIG supports many more languages than I am able to fully support because I am not familiar with the scripting languages.

I have written bindings for Perl and Tcl/Tk. These will be fully supported. I have begun work on Python bindings, but don't know the Python language well enough to full test the work.

15 Tcl/Tk Interface Bindings

Tcl bindings are provided through the interface file cdbtcl.i in the src directory of the cdb distribution. This file is a SWIG interface file that allows SWIG to set up the hooks with the C functions in DCDB.

There are three *typemaps* provided in cdbtcl.i (for an explanation of typemaps, see the SWIG documentation). These essentially cause all error conditions returned from DCDB functions to cause an exception to be raised. The implication of this is that your tcl script will exit with an exception condition (leaving tables open, etc) if you don't handle exceptions in your Tcl code. Look at at the scripts in (topdir)/tests/tcl for examples of how to use Tcl to do real work with DCDB.

The bindings for Tcl and Tk are identical and are both documented in this portion of the documentation.

16 DCDB Tcl/Tk Bindings

The Tcl/Tk bindings for DCDB provide simple wrappers for the functions that serve as the core of DCDB. These functions allow the Tcl developer to use the

complete capabilities of DCDB with the simplicity and quick development of Tcl scripting.

Internally, all tables opened in Tcl scripts are maintained in a simple array. Each time a table is accessed via a table handle, the array is searched for the associated "dbTable" item. Random searches through an array like this are inherently inefficient. For the sake of performance, therefore, it is preferable to keep the number of concurrently open tables down to fewer than twenty or so. If more tables are needed concurrently, the developer should consider using a different data structure to store table handles (a hash would provide better performance for more items, for example). Such a change would require a change to the cdb.c file in the DCDB lib directory. There is a limit currently set on the number of concurrently open tables. That limit is given by the MAX_HANDLES constant. Change this constant to increase this or decrease it.

The developer should read the README file for information on creating the "libcdbtcl.so" and "libconttcl.so" shared objects. Both of these are needed to execute the test scripts in Tcl.

All of the Tcl bindings for DCDB require arguments except dbexit. If the command is entered without an argument, a "usage" string is returned and an error is raised. Care should be taken to insure that production tables are not corrupted during script development.

The DCDB Tcl commands that require a table handle all operate on an open table. If an error occurs, it is possible that the script will abort without cleaning up any open tables. To avoid that possibility in production code, the developer should call the commands in a Tcl "catch" construct. This will allow the script to call "dbexit" before terminating to properly close all open tables and indexes. *Failure to do so will result in data corruption in your indexes and possibly your tables.*

See the sample scripts for examples of how to protect your tables while manipulating them. The following documentation should be considered reference material only. It *will not* show you how to use the Tcl bindings to do real work with DCDB. You should study the test scripts and use them as examples of how to write code using the DCDB bindings and Tcl.

16.1 dbcreate

dbcreate definitionfile

dbcreate creates the tables and workspaces defined in the definition file given by "definitionfile". If an error occurs, an error is raised and a string explaining the error is returned. If dbcreate is successful, "OK" is returned.

For examples of using dbcreate, look at the scripts in (topdir)/tests/tcl. Basically, all of these scripts do the following:

- open a file (flogadd1.df, for example),
- write the code into the file to create the needed tables and indexes,
- close the file,

- call dbcreate with the file name as an argument (“dbcreate flogadd1.df”, for example),
- and remove the .df file

This is the simplest way to insure that the tables and indexes needed by your script are created.

16.2 dbopen

set table [dbopen tablename]

dbopen opens the table given by “tablename”. It is an error if the table doesn't exist. Upon success, dbopen returns a table handle (really, just the table name) that is used in subsequent DCDB Tcl commands. This value should be stored in a variable for further use. If an error occurs, dbopen raises an error and returns a string that describes the error. If other tables are open, the developer should execute the dbopen in a Tcl “catch” statement to allow the script to properly close the other open tables before exiting.

16.3 dbclose

dbclose $tablehandle

dbclose closes the table given by “tablehandle”. It is an error if the table is not open. Upon successful completion, a table and all it's associated indexes will be properly flushed to disk and closed and “OK” is returned. If an error occurs, dbclose raises an error and returns a string that describes the error.

It's actually more common to use dbexit to close all open tables than to close tables individually.

16.4 dbnumrecs

set numrecs [dbnumrecs $tablehandle]

dbnumrecs returns the number of records currently being managed in the table. It is an error if the table is not open yet. Upon successful completion, dbnumrecs returns the number of records. If an error occurs, an exception is raised and a string is returned that describes the error.

16.5 dbnumfields

set numfieds [dbnumfields $tablehandle]

dbnumfields returns the number of fields in the table. It is an error if the table is not open yet. Upon successful completion, dbnumfields returns the number of fields. If an error occurs, an exception is raised and a string is returned that describes the error.

16.6 dbseq

set seq [dbseq $tablehandle]

dbseq returns the "nextSequence" value stored in the table header of the table given by "tablehandle", and then increments the "nextSequence" value. It is an error if the table is not open. Upon successful completion, dbseq returns the sequence value stored in the table before the call was made. If an error occurs, an error is raised and a string describing the error is returned.

16.7 dbiseof

set status [dbiseof $tablehandle]

dbiseof returns "1" if the end of file indicator for the table is TRUE and "0" if it is FALSE. It is an error if the table is not open. If an error occurs, an error is raised and a string is returned that describes the error.

16.8 dbisbof

set status [dbisbof $tablehandle]

dbisbof returns "1" if the beginning of file indicator for the table is TRUE and "0" if it is FALSE. It is an error if the table is not open. If an error occurs, an error is raised and a string is returned that describes the error.

16.9 dbshow

set fldval [dbshow $tablehandle field]

dbshow returns the contents of "field" if successful. If dbretrieve wasn't called before the call to dbshow, the return value will be meaningless. It is an error if the table is not open or "field" is not the name of a valid field in the table. If an error occurs, an error is raised and a string describing the error is returned.

16.10 dbflen

set fldlen [dbflen $tablehandle field]

dbflen returns the length of "field" for the table. It is an error if the table is not open or "field" is not a valid field name. If an error occurs, an error is raised and a string describing the error is returned.

16.11 dbdeclen

set declen [dbdeclen $tablehandle field]

dbdeclen returns the length of decimal portion for "field". If the field is not numeric, this will always be 0. It is an error if the table is not open or "field" is not a valid field name. If an error occurs, an error is raised and a string describing the error is returned.

16.12 dbsetint

set status [dbsetint $tablehandle field intval]

dbsetint sets the tbl->fields[] value for "field" to "intval". The table must be open and "field" must be a valid numerical field. Upon successful completion, the field is set and "OK" is returned. If an error occurs, an error is raised and a string describing the error is returned.

16.13 dbsetnum

set status [dbsetnum $tablehandle field dblval]

dbsetnum works as dbsetint except it accepts double values, as well. If an error occurs, an error is raised and a string describing the error is returned.

16.14 dbsetlog

set status [dbsetlog $tablehandle field logval]

dbsetlog is used to set a logical field in the tbl->fields[] array. Other than that, it functions identically to dbsetint. If an error occurs, an error is raised and a string describing the error is returned.

16.15 dbsetchar

set status [dbsetchar $tablehandle field charval]

dbsetchar is used to set character fields to a value. Other than that, it works identically to dbsetint. If an error occurs, an error is raised and a string describing the error is returned.

16.16 dbsetdate

set status [dbsetdate $tablehandle field dateval]

dbsetdate is used to set the value of date fields. Other than that, it works identically to dbsetint. If an error occurs, an error is raised and a string describing the error is returned.

16.17 dbsettime

set status [dbsettime $tablehandle field timeval]

dbsettime is used to set the value of a time stamp field. Other than that, it works identically to dbsetint. If an error occurs, an error is raised and a string describing the error is returned.

16.18 dbadd

set status [dbadd $tablehandle]

dbadd adds the data placed in the tbl->fields[] array using the dbsetxxx calls to the table. It is an error if a unique index constraint would be violated

by the add. It is an error if the table is not open. Upon successful completion, the data will have been added to the table. If an error occurs, an error is raised and a string describing the error is returned.

16.19 dbretrieve

set status [dbretrieve $tablehandle]

dbretrieve retrieves the data for the current record into the tbl->fields[] array where it is accessible via calls to dbshow. It is an error if the table is not open. If the current record is marked for deletion, "0" is returned. If an error occurs, an error is raised and a string describing the error is returned.

16.20 dbupdate

set status [dbupdate $tablehandle]

dbupdate replaces the contents of the current record with the values stored in the tbl->fields[] array. It is an error if the table is not open. If an error occurs, an error is raised and a string describing the error is returned.

16.21 dbdel

set status [dbdel $tablehandle]

dbdel marks the current record in the table for deletion and returns "OK" if successful. The record is not really deleted until dbpack is run on the table, at which point the deletion is irreversible. It is an error if the table is not open. If an error occurs, an error is raised and a string describing the error is returned.

16.22 cdbundel

set status [dbundel $tablehandle]

dbundel unmarks the current record for deletion, if it is marked. If it is not marked, dbundel does nothing. It is an error if the table is not open. If an error occurs, an error is raised and a string describing the error is returned.

16.23 dbisdeleted

set isdel [dbisdeleted $tablehandle]

dbisdeleted returns "1" if the current record in the table is marked for deletion, "0" otherwise. It is an error if the table is not open. If an error occurs, an error is raised and a string describing the error is returned.

16.24 dbfldname

set fldnm [dbfldname $tablehandle fldnum]

dbfldname returns the name of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum

is less than 0 or greater than the number of fields, an error is raised. An error is likewise raised if there are no open tables, if the file handle is invalid or on some other critical error.

16.25 dbfldtype

set fldtype [dbfldtype $tablehandle fldnum]

dbfldtype returns the type of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is raised. An error is likewise raised if there are no open tables, if the file handle is invalid or on some other critical error.

16.26 dbfldlen

set fldlen [dbfldlen $tablehandle fldnum]

dbfldlen returns the length of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is raised. An error is likewise raised if there are no open tables, if the file handle is invalid or on some other critical error.

16.27 dbflddec

set flddec [dbflddec $tablehandle fldnum]

dbflddec returns the length of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is raised. An error is likewise raised if there are no open tables, if the file handle is invalid or on some other critical error.

16.28 dbcurrent

set status [dbcurrent $tablehandle indexname]

dbcurrent sets the current index for the table to “indexname” and returns “OK” on success. It is an error if the table is not open or if “indexname” is not a valid index for the table. If an error occurs, an error is raised and a string describing the error is returned.

16.29 dbgo

set recno [dbgo tablehandle recno]

dbgo sets the current record to the record numbered “recno” (a 32-bit integer value) on success and returns the record number of the current record. It is an error if the table is not open or “recno” is less than one or greater than the number of records. If an error occurs, an error is raised and a string describing the error is returned.

16.30 dbnext

set status [dbnext $tablehandle]

dbnext sets the current record in the table to the next physical record. It is an error if the table is not open. If the current record is already the last record when dbnext is called, dbnext sets the end of file marker on the table to TRUE (which can be checked with a call to dbiseof). If an error occurs, an error is raised and a string describing the error is returned.

16.31 dbnextindex

set status [dbnextindex $tablehandle]

dbnextindex sets the current record in the table to the next record by the current index. It is an error if the table is not open. If there is no current index, dbnextindex decays to a call to dbnext. If the current record is already the last record when dbnextindex is called, dbnextindex sets the end of file marker on the table to TRUE (which can be checked with a call to dbiseof). If an error occurs, an error is raised and a string describing the error is returned.

16.32 dbprev

set status [dbprev $tablehandle]

dbprev sets the current record in the table to the previous physical record. It is an error if the table is not open. If the current record is already the first record when dbprev is called, dbprev sets the beginning of file marker on the table to TRUE (which can be checked with a call to dbisbof). If an error occurs, an error is raised and a string describing the error is returned.

16.33 dbprevindex

set status [dbprevindex $tablehandle]

dbprevindex sets the current record in the table to the previous record by the current index. It is an error if the table is not open. If the current index is not set, dbprevindex decays to a call to dbprev. If the current record is already the first record when dbprevindex is called, dbprevindex sets the beginning of file marker on the table to TRUE (which can be checked with a call to dbisbof). If an error occurs, an error is raised and a string describing the error is returned.

16.34 dbhead

set status [dbhead $tablehandle]

dbhead sets the current record in the table to the first physical record. It is an error if the table is not open. dbhead clears the BOF and EOF markers for the table (sets them to false). If an error occurs, an error is raised and a string describing the error is returned.

16.35 dbheadindex

set status [dbheadindex $tablehandle]

dbheadindex sets the current record in the table to the first record by the current index. It is an error if the table is not open. If there is no current index set, dbheadindex decays to a call to dbhead. dbheadindex clears the BOF and EOF markers for the table (sets them to false). If an error occurs, an error is raised and a string describing the error is returned.

16.36 dbtail

set status [dbtail $tablehandle]

dbtail sets the current record in the table to the last physical record. It is an error if the table is not open. dbtail clears the BOF and EOF markers for the table (sets them to false). If an error occurs, an error is raised and a string describing the error is returned.

16.37 dbtailindex

set status [dbtailindex $tablehandle]

dbtailindex sets the current record in the table to the last record by the current index. It is an error if the table is not open. If there is no current index, dbtailindex decays to a call to dbtail. dbtailindex clears the BOF and EOF markers for the table (sets them to false). If an error occurs, an error is raised and a string describing the error is returned.

16.38 dbsearchindex

set status [dbsearchindex $tablehandle fieldvalue]

dbsearchindex searches using the current index for the value given by "fieldvalue". dbsearchindex does a best fit search and only returns "0" if an error occurs. It is an error if the table is not open or it doesn't have a current index set. If an error occurs, an error is raised and a string describing the error is returned.

16.39 dbsearchexact

set status [dbsearchexact $tablehandle fieldvalue]

dbsearchexact searches using the current index for the value given by "fieldvalue". dbsearchexact does an exact search and returns "0" if the item is not found. It is an error if the table is not open or it doesn't have a current index set. If an error occurs, an error is raised and a string describing the error is returned.

Note: You should take extra measures when searching for numerical data based on how it is stored. See test3.tcl for a discussion of this issue.

16.40 dbpack

set status [dbpack $tablehandle]

dbpack packs the open table given by the handle and returns "OK" if successful. If an error occurs, an error is raised and a string describing the error is returned.

16.41 dbreindex

set status [dbreindex tablename]

dbreindex reindexes the table given by tablename and returns "OK" if successful. It is an error if the table *is* open. If an error occurs, an error is raised and a string describing the error is returned.

16.42 dbexit

set status [dbexit]

dbexit is the only DCDB Tcl command that does not take any arguments. To get a usage string, simply use "dbexit help". If there are no open tables, it simply returns "OK". If there are open tables, it closes all of them, cleans up the list used to store table handles and returns "OK".

16.43 md5sum

set md5val [md5sum fspec]

md5sum will get an md5 hash on the file given by fspec. If the user doesn't have the permission to open the file or an error occurs, 0 is returned and an error is raised.

16.44 sha1sum

set sha1val [sha1sum fspec]

sha1sum will get an sha1 hash on the file given by fspec. If the user doesn't have the permission to open the file or an error occurs, 0 is returned and an error is raised.

16.45 rmd160sum

set rmdval [rmd160sum fspec]

rmd160sum will get an rmd160 hash on the file given by fspec. If the user doesn't have the permission to open the file or an error occurs, 0 is returned and an error is raised.

16.46 dbtime

set clock1 [dbtime]

dbtime will return a double value that is a fairly high resolution timing value (if your architecture supports it). It uses the elapsed() function defined in btime.c.

16.47 dbnummidx

set number [dbnummidx $table]

dbnummidx returns the number of multi-field indexes in the table given by tablehandle. It is not an error if there are no multi-field indexes for the table.

16.48 dbismidx

set ismultiindexes [dbismidx $table]

dbismidx returns TRUE if there are multi-field indexes associated with the table given by tablehandle table. It returns FALSE otherwise.

16.49 dbmidxname

set fieldname [dbmidxname $table $num]

dbmidxname returns the name of the num'th multi-field index for the table given by tablehandle. It is an error if there are no multi-field indexes for the table. Use dbismidx first to determine if there are multi-field indexes for the table.

16.50 dbmidxblksz

set blksize [dbmidxblksz $table $midxname]

dbmidxblksz returns the blocksize of the multi-field index midxname attached to the table given by tablehandle table. It is an error if the table has no multi-field indexes or if it has none called midxname.

16.51 dbmidxnumfldnames

set numfldnms [dbmidxnumfldnames $table $midxname]

dbmidxnumfldnames returns the number of fields that are used to generate the multi-field index midxname which is attached to the table given by tablehandle table. It is an error if the table has no multi-field indexes attached to it or if it has none called midxname.

16.52 dbmidxfldname

set fldname [dbmidxfldname $table $midxname $num]

dbmidxfldname returns the num'th field name that makes up the multi-index midxname which is attached to the table given by tablehandle table. It is an

error if the table has no multi-field indexes attached to it or if it has none named midxname. It is also an error if the multi-field index given by midxname doesn't have num fields. You should call dbmidxnumfldnames to determine how many field names are used in generating the multi-field index.

16.53 dbisindexed

set isindexed [dbisindexed $table $fieldname]

dbisindexed returns TRUE if the field name fieldname in the table given by the tablehandle table is indexed. It returns FALSE otherwise.

16.54 dbidxblksz

set blksize [dbidxblksz $table $fieldname]

dbidxblksz returns the block size of the index generated for the field fieldname in the table given by the tablehandle table. It is an error if the field is not indexed. You should call dbisindexed to insure that the field is indexed before you call dbidxblksz.

16.55 dbshowinfo

set tableinfo [dbshowinfo $table]

dbshowinfo returns the contents of the info string that is stored in the table header. This is specific to the application and has no meaning as far as DCDB is concerned. It can be a string of up to 125 characters and can contain any value the application programmer desires. It is set with the "info" portion of the "create" command in a .df file.

16.56 dbteststring

set LName [dbteststring 0 64]

dbteststring returns a psuedo-random string of characters, all lower-case. It takes 2 arguments, both integers. If the first argument is set, the second must be 0. Conversely, if the second argument is set, the first must be 0. Setting the first argument gives you a string of that length. For example, "dbteststring 10 0" will return a 10 character string. Setting the second argument will return a string of somewhat random length between 5 and the length specified. So, "dbteststring 0 10" gives you a random string of lower-case letters between 5 and 10 digits long. It is an error if both arguments are 0 or if both are non-zero.

16.57 dbtestupperstring

set LName [dbtestupperstring 0 64]

dbtestupperstring works the same as dbteststring except that the string returned will consist of upper-case alphabetical digits.

16.58 dbtestmixedstring

set teststr [dbtestmixedstring 0 35]

dbtestmixedstring works the same as dbteststring except that the string returned will be of mixed upper and lower-case alphabetical digits.

16.59 dbtestnumber

set SSN [dbtestnumber 9 0]

dbtestnumber works the same as dbteststring except that it returns a string of numerical digits.

16.60 dbtestnumstring

set LicensePlate [dbtestnumstring 7 0]

dbtestnumstring works the same as dbteststring except that it returns a string of combined numercal and upper-case alphabetical digits.

16.61 dbencrypt

set encrypted [dbencrypt $data $size $password]

dbencrypt encrypts the data in the data argument using the password in the passwd argument. The size argument should be a multiple of 8, although if it is not, dbencrypt will find the closest multiple of 8 greater than size. The maximum size for a data item is 2048 bytes. The maximum length of the password is 56 bytes (448 bits).

dbencrypt uses the blowfish algorithm. This algorithm is unencumbered with patents and *restrictive* copyrights. It is an open source implementation written by Bruce Schneier (thanks, Bruce). You can use dbencrypt to create and manage passwords or to encrypt data in blocks.

16.62 dbdecrypt

set decrypted [dbdecrypt $data $size $passwd]

dbdecrypt decrypts the data given by the data argument using the password given by passwd. The data argument should contain data that was encrypted with dbencrypt and the passwd argument should have the same password used to encrypt the data. The size argument should also be the size of the data before it was encrypted.

16.63 crc32sum

set crcsum [crc32sum $fname]

crc32sum will generate the crc32 check sum of the file given by fname and return that value. This is the 32 bit check sum code from (LINUX_SRC)/fs/jffs2/crc32* from the 2.4.18 kernel modified for my my use here.

16.64 bcnumadd

set total [bcnumadd 240.37 23.28 2]
bcnumadd uses the number engine from bc to add the two numbers.

16.65 bcnumsub

set total [bcnumsub 240.37 23.28 2]
bcnumsub uses the number engine from bc to subtract the two numbers.

16.66 bcnumcompare

set result [bcnumcompare $num1 $num2 2]
bcnumcompare uses the number engine from bc to compare the two numbers.

16.67 bcnummultiply

set total [bcnummultiply 240.37 -23.28 2]
bcnummultiply uses the number engine from bc to multiply the two numbers.

16.68 bcnumdivide

set total [bcnumdivide 240.37 -23.28 2]
bcnumdivide uses the number engine from bc to divide the two numbers.

16.69 bcnumraise

set total [bcnumraise 240.37 -23.28 2]
bcnumraise uses the number engine from bc to raise the fist number to the power of the second number.

16.70 bcnumiszero

set status [bcnumiszero $result]
bcnumiszero uses the GNU bc number engine to determine if arg1 is zero and returns 1 if it is and 0 if it is not.

16.71 bcnumisnearzero

set status [bcnumisnearzero $result 2]
bcnumisnearzero uses the GNU bc number engine to determine if arg1 is near zero. It returns 1 if it is and 0 if it is not.

16.72 bcnumisneg

set status [bcnumisneg $result]
bcnumisneg uses the GNU bc number engine to determine if arg1 is negative. It returns 1 if it is and 0 if it is not.

16.73 bcnumuninit

bcnumuninit

bcnumuninit de-initializes the bcnum stuff (basically, freeing some bcnum numbers that are preallocated). This should be called when you are done with the bcnum bindings. If you don't do it manually, there is an atexit() function that will do it for you. This only returns _OK_.

17 Perl Interface Bindings

Perl bindings are provided through the interface file cdbpl.i in the src directory of the cdb distribution. This file is a SWIG interface file that allows SWIG to set up the hooks with the C functions in DCDB.

There are three *typemaps* provided in cdbpl.i (for an explanation of typemaps, see the SWIG documentation). These essentially cause all error conditions returned from DCDB functions to cause an exception to be raised. The implication of this is that your perl script will exit with an exception condition (leaving tables open, etc) if you don't handle exceptions in your perl code. Look at test[1,2].pl to see examples of how this would be handled.

18 DCDB Perl Bindings

The Perl bindings for DCDB provide simple wrappers for the functions that serve as the core of DCDB. These functions allow the perl developer to use the complete capabilities of DCDB with the flexibility and quick development of perl scripting.

Internally, all tables opened in perl scripts are maintained in a simple array. Each time a table is accessed via a table handle, the array is searched for the associated "dbTable" item. Random searches through an array like this are inherently inefficient. For the sake of performance, therefore, it is preferable to keep the number of concurrently open tables down to fewer than ten. If more tables are needed concurrently, the developer should consider using a different data structure to store table handles (a hash would provide better performance for more items, for example). Such a change would require a change to the cdb.c file in the DCDB lib directory. If you want to change the number of tables that can be open concurrently, you can change the defined constant called MAX_HANDLES to allow for as many open tables as you would like.

The developer should read the README file for information on creating the "libcdbperl.so" shared library and the "libcdbperl.pm" perl module.

All of the perl bindings for DCDB require arguments except dbexit. If the command is entered without an argument, a "usage" string is returned and an error is raised. Care should be taken to insure that production tables are not corrupted during script development.

The DCDB perl commands that require a table handle all operate on an open table. If an error occurs, it is possible that the script will abort without

cleaning up any open tables. To avoid that possibility in production code, the developer should call the commands in a perl "eval" construct. This will allow the script to call "dbexit" before terminating to properly close all open tables and indexes. *Failure to do so will result in data corruption in your indexes and possibly your tables.*

See the sample scripts for examples of how to protect your tables while manipulating them. You can find them at (topdir)/extras/perl. The following documentation should be considered reference material only. It *will not* show you how to use the Perl bindings to do real work with DCDB. You should study the sample scripts and use them as examples of how to write code using the DCDB bindings and Perl.

18.1 dbcreate

&cdbperl::dbcreate ("definitionfile");

dbcreate creates the tables and workspaces defined in the definition file given by "definitionfile". If an error occurs, an error is raised and a string explaining the error is returned. If dbcreate is successful, "OK" is returned.

For examples of using dbcreate, look at the scripts in extras/perl. Basically, all of these scripts do the following:

- open a file (flogadd1.df, for example),
- write the code into the file to create the needed tables and indexes,
- close the file,
- call dbcreate with the file name as an argument
- and remove the .df file

This is the simplest way to insure that the tables and indexes needed by your script are created.

18.2 dbopen

$table = &cdbperl::dbopen ("tablename");

dbopen opens the table given by "tablename". It is an error if the table doesn't exist. Upon success, dbopen returns a table handle (really, just the table name) that is used in subsequent DCDB perl commands. This value should be stored in a variable for further use. If an error occurs, dbopen raises an error and returns a string that describes the error. If other tables are open, the developer should execute the dbopen in a perl "eval" statement to allow the script to properly close the other open tables before exiting.

18.3 dbclose

&cdbperl::dbclose ($tablehandle);

dbclose closes the table given by "tablehandle". It is an error if the table is not open. Upon successful completion, a table and all it's associated indexes will be properly flushed to disk and closed and "OK" is returned. If an error occurs, dbclose raises an error and returns a string that describes the error.

It's actually more common to use dbexit to close all open tables than to close tables individually.

18.4 dbnumrecs

$numrecs = &cdbperl::dbnumrecs ($tablehandle);

dbnumrecs returns the number of records currently being managed in the table. It is an error if the table is not open yet. Upon successful completion, dbnumrecs returns the number of records. If an error occurs, an exception is raised and a string is returned that describes the error.

18.5 dbnumfields

$numflds = &cdbperl::dbnumfields ($tablehandle);

dbnumfields returns the number of fields in the table. It is an error if the table is not open yet. Upon successful completion, dbnumfields returns the number of fields. If an error occurs, an exception is raised and a string is returned that describes the error.

18.6 dbseq

$seq = &cdbperl::dbseq ($tablehandle);

dbseq returns the "nextSequence" value stored in the table header of the table given by "tablehandle", and then increments the "nextSequence" value. It is an error if the table is not open. Upon successful completion, dbseq returns the sequence value stored in the table before the call was made. If an error occurs, an error is raised and a string describing the error is returned.

18.7 dbiseof

$status = &cdbperl::dbiseof ($tablehandle);

dbiseof returns "1" if the end of file indicator for the table is TRUE and "0" if it is FALSE. It is an error if the table is not open. If an error occurs, an error is raised and a string is returned that describes the error.

18.8 dbisbof

$status = &cdbperl::dbisbof ($tablehandle);

dbisbof returns “1” if the beginning of file indicator for the table is TRUE and “0” if it is FALSE. It is an error if the table is not open. If an error occurs, an error is raised and a string is returned that describes the error.

18.9 dbshow

$fldval = &cdbperl::dbshow ($tablehandle, “field”);

dbshow returns the contents of “field” if successful. If dbretrieve wasn’t called before the call to dbshow, the return value will be meaningless. It is an error if the table is not open or “field” is not the name of a valid field in the table. If an error occurs, an error is raised and a string describing the error is returned.

18.10 dbflen

$fldlen = &cdbperl::dbflen ($tablehandle, “field”);

dbflen returns the length of “field” for the table. It is an error if the table is not open or “field” is not a valid field name. If an error occurs, an error is raised and a string describing the error is returned.

18.11 dbdeclen

$declen = &cdbperl::dbdeclen ($tablehandle, “field”);

dbdeclen returns the length of decimal portion for “field”. If the field is not numeric, this will always be 0. It is an error if the table is not open or “field” is not a valid field name. If an error occurs, an error is raised and a string describing the error is returned.

18.12 dbsetint

$status = &cdbperl::dbsetint ($tablehandle, “field”, intval);

dbsetint sets the tbl->fields[] value for “field” to “intval”. The table must be open and “field” must be a valid numerical field. Upon successful completion, the field is set and “OK” is returned. If an error occurs, an error is raised and a string describing the error is returned.

18.13 dbsetnum

$status = &cdbperl::dbsetnum ($tablehandle, “field”, dblval);

dbsetnum works as dbsetint except it accepts double values, as well. If an error occurs, an error is raised and a string describing the error is returned.

18.14 dbsetlog

$status = &cdbperl::dbsetlog ($tablehandle, “field”, ‘logval’);

dbsetlog is used to set a logical field in the tbl->fields[] array. Other than that, it functions identically to dbsetint. If an error occurs, an error is raised and a string describing the error is returned.

18.15 dbsetchar

$status = &cdbperl::dbsetchar ($tablehandle, "field", "charval");

dbsetchar is used to set character fields to a value. Other than that, it works identically to dbsetint. If an error occurs, an error is raised and a string describing the error is returned.

18.16 dbsetdate

$status = &cdbperl::dbsetdate ($tablehandle, "field", "dateval");

dbsetdate is used to set the value of date fields. Other than that, it works identically to dbsetint. If an error occurs, an error is raised and a string describing the error is returned.

18.17 dbsettime

$status = &cdbperl::dbsettime ($tablehandle, "field", "timeval");

dbsettime is used to set the value of a time stamp field. Other than that, it works identically to dbsetint. If an error occurs, an error is raised and a string describing the error is returned.

18.18 dbadd

$status = &cdbperl::dbadd ($tablehandle);

dbadd adds the data placed in the tbl->fields[] array using the dbsetxxx calls to the table. It is an error if a unique index constraint would be violated by the add. It is an error if the table is not open. Upon successful completion, the data will have been added to the table. If an error occurs, an error is raised and a string describing the error is returned.

18.19 dbretrieve

$status = &cdbperl::dbretrieve ($tablehandle);

dbretrieve retrieves the data for the current record into the tbl->fields[] array where it is accessible via calls to dbshow. It is an error if the table is not open. If the current record is marked for deletion, "0" is returned. If an error occurs, an error is raised and a string describing the error is returned.

18.20 dbupdate

$status = &cdbperl::dbupdate ($tablehandle);

dbupdate replaces the contents of the current record with the values stored in the tbl->fields[] array. It is an error if the table is not open. If an error occurs, an error is raised and a string describing the error is returned.

18.21 dbdel

$status = &cdbperl::dbdel ($tablehandle);

dbdel marks the current record in the table for deletion and returns "OK" if successful. The record is not really deleted until dbpack is run on the table, at which point the deletion is irreversible. It is an error if the table is not open. If an error occurs, an error is raised and a string describing the error is returned.

18.22 cdbundel

$status = &cdbperl::dbundel ($tablehandle);

dbundel unmarks the current record for deletion, if it is marked. If it is not marked, dbundel does nothing. It is an error if the table is not open. If an error occurs, an error is raised and a string describing the error is returned.

18.23 dbisdeleted

$status = &cdbperl::dbisdeleted ($tablehandle);

dbisdeleted returns "1" if the current record in the table is marked for deletion, "0" otherwise. It is an error if the table is not open. If an error occurs, an error is raised and a string describing the error is returned.

18.24 dbfldname

$fldnm = &cdbperl::dbfldname ($tablehandle, fldnum);

dbfldname returns the name of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is raised. An error is likewise raised if there are no open tables, if the file handle is invalid or on some other critical error.

18.25 dbfldtype

$fldtype = &cdbperl::dbfldtype ($tablehandle, fldnum);

dbfldtype returns the type of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is raised. An error is likewise raised if there are no open tables, if the file handle is invalid or on some other critical error.

18.26 dbfldlen

$fldlen = &cdbperl::dbfldlen ($tablehandle, fldnum);

dbfldlen returns the length of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is raised. An error is likewise raised if there are no open tables, if the file handle is invalid or on some other critical error.

18.27 dbflddec

$flddec = &cdbperl::dbfldlen ($tablehandle, fldnum);

dbflddec returns the length of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is raised. An error is likewise raised if there are no open tables, if the file handle is invalid or on some other critical error.

18.28 dbcurrent

$status = &cdbperl::dbcurrent ($tablehandle, “indexname”);

dbcurrent sets the current index for the table to “indexname” and returns “OK” on success. It is an error if the table is not open or if “indexname” is not a valid index for the table. If an error occurs, an error is raised and a string describing the error is returned.

18.29 dbgo

$recno = &cdbperl::dbgo ($tablehandle, recno);

dbgo sets the current record to the record numbered “recno” (a 32-bit integer value) on success and returns the record number of the current record. It is an error if the table is not open or “recno” is less than one or greater than the number of records. If an error occurs, an error is raised and a string describing the error is returned.

18.30 dbnext

$status = &cdbperl::dbnext ($tablehandle);

dbnext sets the current record in the table to the next physical record. It is an error if the table is not open. If the current record is already the last record when dbnext is called, dbnext sets the end of file marker on the table to TRUE (which can be checked with a call to dbiseof). If an error occurs, an error is raised and a string describing the error is returned.

18.31 dbnextindex

$status = &cdbperl::dbnextindex ($tablehandle);

dbnextindex sets the current record in the table to the next record by the current index. It is an error if the table is not open. If there is no current index, dbnextindex decays to a call to dbnext. If the current record is already the last record when dbnextindex is called, dbnextindex sets the end of file marker on the table to TRUE (which can be checked with a call to dbiseof). If an error occurs, an error is raised and a string describing the error is returned.

18.32 dbprev

$status = &cdbperl::dbprev ($tablehandle);

dbprev sets the current record in the table to the previous physical record. It is an error if the table is not open. If the current record is already the first record when dbprev is called, dbprev sets the beginning of file marker on the table to TRUE (which can be checked with a call to dbisbof). If an error occurs, an error is raised and a string describing the error is returned.

18.33 dbprevindex

$status = &cdbperl::dbprevindex ($tablehandle);

dbprevindex sets the current record in the table to the previous record by the current index. It is an error if the table is not open. If the current index is not set, dbprevindex decays to a call to dbprev. If the current record is already the first record when dbprevindex is called, dbprevindex sets the beginning of file marker on the table to TRUE (which can be checked with a call to dbisbof). If an error occurs, an error is raised and a string describing the error is returned.

18.34 dbhead

$status = &cdbperl::dbhead ($tablehandle);

dbhead sets the current record in the table to the first physical record. It is an error if the table is not open. dbhead clears the BOF and EOF markers for the table (sets them to false). If an error occurs, an error is raised and a string describing the error is returned.

18.35 dbheadindex

$status = &cdbperl::dbheadindex ($tablehandle);

dbheadindex sets the current record in the table to the first record by the current index. It is an error if the table is not open. If there is no current index set, dbheadindex decays to a call to dbhead. dbheadindex clears the BOF and EOF markers for the table (sets them to false). If an error occurs, an error is raised and a string describing the error is returned.

18.36 dbtail

$status = &cdbperl::dbtail ($tablehandle);

dbtail sets the current record in the table to the last physical record. It is an error if the table is not open. dbtail clears the BOF and EOF markers for the table (sets them to false). If an error occurs, an error is raised and a string describing the error is returned.

18.37 dbtailindex

$status = &cdbperl::dbtailindex ($tablehandle);

dbtailindex sets the current record in the table to the last record by the current index. It is an error if the table is not open. If there is no current index, dbtailindex decays to a call to dbtail. dbtailindex clears the BOF and EOF markers for the table (sets them to false). If an error occurs, an error is raised and a string describing the error is returned.

18.38 dbsearchindex

$status = &cdbperl::dbsearchindex ($tablehandle, "fieldvalue");

dbsearchindex searches using the current index for the value given by "fieldvalue". dbsearchindex does a best fit search and only returns "0" if an error occurs. It is an error if the table is not open or it doesn't have a current index set. If an error occurs, an error is raised and a string describing the error is returned.

18.39 dbsearchexact

$status = &cdbperl::dbsearchexact ($tablehandle, "fieldvalue");

dbsearchexact searches using the current index for the value given by "fieldvalue". dbsearchexact does an exact search and returns "0" if the item is not found. It is an error if the table is not open or it doesn't have a current index set. If an error occurs, an error is raised and a string describing the error is returned.

Note: You should take extra measures when searching for numerical data based on how it is stored. See test3.pl for a discussion of this issue.

18.40 dbpack

$status = &cdbperl::dbpack ($tablehandle);

dbpack packs the open table given by the handle and returns "OK" if successful. If an error occurs, an error is raised and a string describing the error is returned.

18.41 dbreindex

$status = &cdbperl::dbreindex (“tablename”);

dbreindex reindexes the table given by tablename and returns “OK” if successful. It is an error if the table *is* open. If an error occurs, an error is raised and a string describing the error is returned.

18.42 dbexit

$status = &cdbperl::dbexit;

dbexit is the only DCDB Tcl command that does not take any arguments. To get a usage string, simply use “dbexit help”. If there are no open tables, it simply returns “OK”. If there are open tables, it closes all of them, cleans up the list used to store table handles and returns “OK”.

18.43 md5sum

$md5val = &cdbperl::md5sum (“fspec”);

md5sum will get an md5 hash on the file given by fspec. If the user doesn't have the permission to open the file or an error occurs, 0 is returned and an error is raised. The return value is meaningless.

18.44 sha1sum

$sha1val = &cdbperl::sha1sum (“fspec”);

sha1sum will get an sha1 hash on the file given by fspec. If the user doesn't have the permission to open the file or an error occurs, 0 is returned and an error is raised. The return value is meaningless.

18.45 rmd160sum

$rmdval = &cdbperl::rmd160sum (“fspec”);

rmd160sum will get an rpm160 hash on the file given by fspec. If the user doesn't have the permission to open the file or an error occurs, 0 is returned and an error is raised. The return value is meaningless.

18.46 dbtime

set clock1 [dbtime]

dbtime will return a double value that is a fairly high resolution timing value (if your architecture supports it). It uses the elapsed() function defined in btime.c.

18.47 dbnummidx

$number = &cdbperl::dbnummidx ($tablehandle);

dbnummidx returns the number of multi-field indexes in the table given by tablehandle. It is an error if there are no multi-field indexes for the table.

18.48 dbmidxname

$fieldname = &cdbperl::dbmidxname ($table, $num);

dbmidxname returns the name of the num'th multi-field index for the table given by tablehandle. It is an error if there are no multi-field indexes for the table. Use dbismidx first to determine if there are multi-field indexes for the table.

18.49 dbmidxname

$fieldname = &cdbperl::dbmidxname ($table, $num);

dbmidxname returns the name of the num'th multi-field index for the table given by tablehandle. It is an error if there are no multi-field indexes for the table. Use dbismidx first to determine if there are multi-field indexes for the table.

18.50 dbmidxblksz

$blksize = dbmidxblksz ($table, $midxname);

dbmidxblksz returns the blocksize of the multi-field index midxname attached to the table given by tablehandle table. It is an error if the table has no multi-field indexes or if it has none called midxname.

18.51 dbmidxnumfldnames

$numfldnms = &cdbperl::dbmidxnumfldnames ($table, $midxname);

dbmidxnumfldnames returns the number of fields that are used to generate the multi-field index midxname which is attached to the table given by tablehandle table. It is an error if the table has no multi-field indexes attached to it or if it has none called midxname.

18.52 dbmidxfldname

$fldname = &cdbperl::dbmidxfldname ($table, $midxname, $num);

dbmidxfldname returns the num'th field name that makes up the multi-index midxname which is attached to the table given by tablehandle table. It is an error if the table has no multi-field indexes attached to it or if it has none named midxname. It is also an error if the multi-field index given by midxname doesn't have num fields. You should call dbmidxnumfldnames to determine how many field names are used in generating the multi-field index.

18.53 dbisindexed

$isindexed = &cdbperl::dbisindexed ($table, $fieldname);

dbisindexed returns TRUE if the field name fieldname in the table given by the tablehandle table is indexed. It returns FALSE otherwise.

18.54 dbidxblksz

$blksize = &cdbperl::dbidxblksz ($table, $fieldname);

dbidxblksz returns the block size of the index generated for the field fieldname in the table given by the tablehandle table. It is an error if the field is not indexed. You should call dbisindexed to insure that the field is indexed before you call dbidxblksz.

18.55 dbshowinfo

$tableinfo = &cdbperl::dbshowinfo($table);

dbshowinfo returns the contents of the info string that is stored in the table header. This is specific to the application and has no meaning as far as DCDB is concerned. It can be a string of up to 125 characters and can contain any value the application programmer desires. It is set with the "info" portion of the "create" command in a .df file.

18.56 dbteststring

$LName = &cdbperl::dbteststring(0, 64);

dbteststring returns a psuedo-random string of characters, all lower-case. It takes 2 arguments, both integers. If the first argument is set, the second must be 0. Conversely, if the second argument is set, the first must be 0. Setting the first argument gives you a string of that length. For example, "dbteststring(10,0)" will return a 10 character string. Setting the second argument will return a string of somewhat random length between 5 and the length specified. So, "dbteststring(0,10)" gives you a random string of lower-case letters between 5 and 10 digits long. It is an error if both arguments are 0 or if both are non-zero.

18.57 dbtestupperstring

$LName = &cdbperl::dbtestupperstring(0,64);

dbtestupperstring works the same as dbteststring except that the string returned will consist of upper-case alphabetical digits.

18.58 dbtestmixedstring

$teststr = &cdbperl::dbtestmixedstring(0,35);

dbtestmixedstring works the same as dbteststring except that the string returned will be of mixed upper and lower-case alphabetical digits.

18.59 dbtestnumber

$SSN = &cdbperl::dbtestnumber(9,0);

dbtestnumber works the same as dbteststring except that it returns a string of numerical digits.

18.60 dbtestnumstring

$LicensePlateNumber = &cdbperl::dbtestnumstring(7,0);

dbtestnumstring works the same as dbteststring except that it returns a string of combined numercal and upper-case alphabetical digits.

18.61 dbencrypt

$encrypted = &cdbperl::dbencrypt($data,$size,$password);

dbencrypt encrypts the data in the data argument using the password in the passwd argument. The size argument should be a multiple of 8, although if it is not, dbencrypt will find the closest multiple of 8 greater than size. The maximum size for a data item is 2048 bytes. The maximum length of the password is 56 bytes (448 bits).

dbencrypt uses the blowfish algorithm. This algorithm is unencumbered with patents and *restrictive* copyrights. It is an open source implementation written by Bruce Schneier (thanks, Bruce). You can use dbencrypt to create and manage passwords or to encrypt data in blocks.

18.62 dbdecrypt

$decrypted = &cdbperl::dbdecrypt($data,$size,$passwd);

dbdecrypt decrypts the data given by the data argument using the password given by passwd. The data argument should contain data that was encrypted with dbencrypt and the passwd argument should have the same password used to encrypt the data. The size argument should also be the size of the data before it was encrypted.

18.63 crc32sum

$crcsum = $cdbperl::crc32sum($fname);

crc32sum will generate the crc32 check sum of the file given by fname and return that value. This is the 32 bit check sum code from LINUX_SRC/fs/jffs2/crc32* from the 2.4.18 kernel modified for my my use here.

18.64 bcnumadd

$total = &cdbperl::bcnumadd(”240.37”, ”23.28”, 2);

bcnumadd uses the number engine from bc to add the two numbers.

18.65 bcnumsub

$total = &cdbperl::bcnumsub("240.37", "23.28", 2);
bcnumsub uses the number engine from bc to subtract the two numbers.

18.66 bcnumcompare

$result = &cdbperl::bcnumcompare($num1, $num2, 2);
bcnumcompare uses the number engine from bc to compare the two numbers.

18.67 bcnummultiply

$total = &cdbperl::bcnummultiply ("240.37","-23.28", 2);
bcnummultiply uses the number engine from bc to multiply the two numbers.

18.68 bcnumdivide

$total = &cdbperl::bcnumdivide ("240.37", "10.00", 2);
bcnumdivide uses the number engine from bc to divide the two numbers.

18.69 bcnumraise

$total = &cdbperl::bcnumraise("240.37", "-23.28", 2);
bcnumraise uses the number engine from bc to raise the fist number to the power of the second number.

18.70 bcnumiszero

$status = &cdbperl::bcnumiszero ($result);
bcnumiszero uses the GNU bc number engine to determine if arg1 is zero and returns 1 if it is and 0 if it is not.

18.71 bcnumisnearzero

$status = &cdbperl::bcnumisnearzero ($result, 2);
bcnumisnearzero uses the GNU bc number engine to determine if arg1 is near zero. It returns 1 if it is and 0 if it is not.

18.72 bcnumisneg

$status = &cdbperl::bcnumisneg ($result);
bcnumisneg uses the GNU bc number engine to determine if arg1 is negative. It returns 1 if it is and 0 if it is not.

18.73 bcnumuninit

&cdbperl::bcnumuninit();

bcnumuninit de-initializes the bcnum stuff (basically, freeing some bcnum numbers that are preallocated). This should be called when you are done with the bcnum bindings. If you don't do it manually, there is an atexit() function that will do it for you. This only returns _OK_.

19 CINT Bindings

CINT is a C/C++ interpreter that is available for numerous architectures under the GPL license. It is a fairly excellent piece of work that provides a simple interface for embedding libraries like DCDB. The advantage of using CINT is that the development that is done, if done carefully, can be compiled into native binaries using the C/C++ compiler. This is quite an advantage.

The interface provided for CINT is almost identical to that provided for Tcl/Tk and Perl, except for the syntax differences between the languages. The function calls have the same arguments. The primary difference between CINT and Tcl/Tk/Perl is that when an error occurs, there is no exception raised in CINT; an error message is merely returned.

There is a README in the cdb-1.1/src/cint directory which has information on compiling CINT for the supported platforms. The main DCDB README file (cdb-1.1/README) has information on creating a "mycint" binary with the DCDB extensions and the container. There are example scripts (with a .c extension) in cdb-1.1/src/cint/scripts which can be interpreted by CINT or compiled into native binaries.

Error conditions are communicated via function return values. If an error occurs, as indicated by the return value, the variable cdberror will have a description of the error. cdberror is defined as follows:

```
extern char cdberror[];
```

19.1 dbcreate

```
int status;
status = dbcreate("deffile.df");
if (status == -1) {
  Process the error...
}
```

dbcreate creates the tables and workspaces defined in the definition file given by "deffile.df". If any error occurs, an error (-1) is returned. If dbcreate is successful, 0 is returned.

19.2 dbopen

```
#define SIZE 115
char handle[SIZE+1];
char *cp;
...
cp = dbopen ("database.db");
if (0 == cp) {
  Process error
}
strcpy (handle, cp);
```

dbopen opens the table given by "database.db". It is an error if the table doesn't exist. Upon success, dbopen returns a read-only pointer to the name of a handle. That handle is to be used for all other table access functions. This value must be copied to a char array, because many of the access functions use the same static storage space to return results as the array the handle name is returned in. The handle is really just the name of the table, so in a pinch you can replace the handle name with "database.db" (in this example).

If an error occurs, dbopen returns a NULL pointer and cdberror contains a description of the error.

19.3 dbclose

```
int status;
...
status = dbclose ("table.db");
if (status == -1)
  Handle error...
```

dbclose closes the table given by the argument ("table.db" in this example), which is actually a table handle. It is an error if the table is not open. Upon successful completion, a table and all it's associated indexes will be properly flushed to disk and closed and 0 is returned. If an error occurs, dbclose returns -1 and cdberror will contain a read-only string that describes the error.

It is entirely permissible to use dbexit to close all open tables right before exiting, should you choose to do so.

19.4 dbnumrecs

```
int numrecs;
...
numrecs = dbnumrecs ("table.db");
if (numrecs == -1)
  handle the error...
```

dbnumrecs returns the number of records currently being managed in the table. It is an error if the table is not open yet. Upon successful completion, dbnumrecs restores the number of records. If an error occurs, an exception is raised and a string is returned that describes the error.

19.5 dbnumfields

```
int numfields;
...
numfields = dbnumfields ("table.db");
if (numfields == -1)
  handle the error...
```

dbnumfields returns the number of fields in the table given by "table.db" in this example (this is a table handle). It is an error if the table is not open yet. Upon successful completion, dbnumfields returns the number of fields. If an error occurs, dbnumfields returns -1 and cdberror contains a read-only string that describes the error.

19.6 dbseq

```
int seq;
...
seq = dbseq ("table.db");
if (seq == -1)
  handle the error...
```

dbseq returns the "nextSequence" value stored in the table header of the table given by "table.db" (a table handle). It then increments the "nextSequence" value. It is an error if the table is not open. Upon successful completion, dbseq returns the sequence value stored in the table before the call was made. On error, a -1 is returned and cdberror contains a read-only string that describes the error.

19.7 dbiseof

```
dbhead ("table.db");
while (! dbiseof ("table.db")) {
  dbretrieve ("table.db");
  do something with the data...
  dbnext ("table.db");
}
```

dbiseof returns 1 if the end of file indicator is TRUE for the table given by "table.db" (a table handle) and 0 if it is FALSE. It is an error if the table is

not open. On error, -1 is returned and cdberror contains a read-only string that describes the error.

19.8 dbisbof

```
dbtail ("table.db");
while (! dbiseof ("table.db")) {
  dbretrieve ("table.db");
  do something with the data...
  dbprev ("table.db");
}
```

dbisof returns 1 if the beginning of file indicator is TRUE for the table given by "table.db" (a table handle) and 0 if it is FALSE. It is an error if the table is not open. On error, -1 is returned and cdberror contains a read-only string that describes the error.

19.9 dbshow

```
char *field;
...
field = dbshow ("table.db", "SSNumber");
if (field == 0)
  handle the error...
printf ("\nSSNumber in table.db is %s\n", field);
```

dbshow is used to get the value of the field in the table "table.db" (a table handle). The field is given by "SSNumber". If successful, dbshow returns the information in a read-only static char string that is overwritten after each call to dbshow. If there is an error, dbshow returns a NULL pointer and cdberror contains a string that describes the error.

19.10 dbflen

```
int flen;
...
flen = dbflen ("table.db", "SSNumber");
if (flen == -1)
  handle the error...
printf (\nLength of SSNumber in table.db is %d\n", flen);
```

dbflen returns the length of the field "SSNumber" in the table "table.db" (a table handle). If an error occurs, dbflen returns -1 and cdberror is a read-only string that describes the error.

19.11 dbdeclen

```
int declen;
...
declen = dbdeclen ("table.db", "SSNumber");
if (declen == -1)
  handle error here...
```

dbdeclen returns the length of decimal portion for the field given by "SSNumber". If the field is not numeric, this will always be 0. It is an error if the table is not open or "SSNumber" is not a valid field name. If an error occurs, an error is raised and a string describing the error is returned. The argument "table.db" is a table handle.

19.12 dbsetint

```
int status;
....
status = dbsetint ("table.db", "Age", 42);
if (status == -1)
  handle the error
```

dbsetint sets the tbl->fields[] value for the field given by "Age" in the table given by "table.db" (this is a table handle) to "42" in this example. The table must be open and "Age" must be a valid numerical field. Upon successful completion, the field is set and 0 is returned. If an error occurs, -1 is returned and cdberror contains a read-only string describing the error.

19.13 dbsetnum

```
int status;
...
status = dbsetnum ("table.db", "Salary", 95550.00);
if (status == -1)
  handle the error
```

dbsetnum works as dbsetint except it accepts a double value field. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

19.14 dbsetlog

```
int status;
...
status = dbsetlog ("table.db", "isMarried", 'Y');
if (status == -1)
  handle the error
```

dbsetlog is used to set a logical field in the tbl->fields[] array of the table "table.db" (a table handle). Other than that, it functions identically to dbsetint. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

19.15 dbsetchar

```
int status;
...
status = dbsetchar ("table.db", "LName", "May");
if (status == -1)
  handle the error
```

dbsetchar works as dbsetint except it accepts a string value field. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

19.16 dbsetdate

```
int status;
...
status = dbsetdate ("table.db", "dateApplied", "20031105");
if (status == -1)
  handle the error
```

dbsetdate works as dbsetint except it accepts a date value field. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

19.17 dbsettime

```
int status;
...
status = dbsettime ("table.db", "timeApplied", "Wed Nov  5 15:23:40 2003");
if (status == -1)
  handle the error
```

dbsettime works as dbsetint except it accepts a time stamp value. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

19.18 dbadd

```
int status;
...
status = dbadd ("table.db");
if (status == -1) {
  if (! strcmp (cdberror, "adding record: unique index constraint violated"))
    deal with unique constraint violations
```

```
  else
    handle the error
}
```

dbadd adds the data placed in the tbl->fields[] array using the dbsetxxx function calls. It is an error if a unique index constraint would be violated by the add; however, it is application dependent whether this should result in the program exiting. It is an error if the table is not open. Upon successful completion, the data will have been added to the table. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

19.19 dbretrieve

```
int status;
...
status = dbretrieve ("table.db");
if (status == -1)
  handle the error
if (status == 0)
  skip a deleted field
Here, do something with the data retrieved
```

dbretrieve retrieves the data for the current record into the tbl->fields[] array where it is accessible via calls to dbshow. It is an error if the table is not open. If the current record is marked for deletion, "0" is returned. You should be aware that because you move to a record doesn't mean the record is retrieved. You must call dbretrieve after movement functions in order to get the data from the table record. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

19.20 dbdel

```
int status;
status = dbdel ("table.db");
if (status == -1)
  handle the error
```

dbdel marks the current record in the table for deletion and returns 0 if successful. The record is not really deleted until dbpack is run on the table, at which point the deletion is irreversible. It is an error if the table is not open. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

19.21 dbundel

```
int status;
...
status = dbundel ("table.db");
if (status == -1)
  handle the error
```

dbundel unmarks the current record for deletion, if it is marked. If it is not marked, dbundel does nothing. It is an error if the table is not open. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

19.22 dbisdeleted

```
int isdel = 0;
...
isdel = dbisdeleted ("table.db");
if (isdel == -1)
  handle the error
if (isdel)
  dbundel ("table.db");
```

dbisdeleted returns "1" if the current record in the table is marked for deletion, "0" otherwise. It is an error if the table is not open. If an error occurs, -1 is returned and cdberror is a read-only string describing the error.

19.23 dbdbfldname

```
#define SIZE 64
char *cp;
char fldnm[SIZE+1];
cp = dbfldname ("table.db", 0);
if (cp == 0)
  handle the error
strcpy (fldnm, cp);
```

dbfldname returns the name of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is returned. An error is likewise returned if there are no open tables, if the file handle is invalid or on some other critical error. If there is an error, a NULL pointer is returned and cdberror contains a read-only string describing the error.

19.24 dbfldtype

```
#define SIZE 10
```

```
char *cp;
char fldtyp[SIZE+1];
cp = dbfldtype ("table.db", 0);
if (cp == 0)
  handle the error
strcpy (fldtyp, cp);
```

dbfldtype returns the type of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is returned. An error is likewise returned if there are no open tables, if the file handle is invalid or on some other critical error. If there is any error, a NULL pointer is returned and cdberror contains a read-only string describing the error.

19.25 dbfldlen

```
int fldlen;
...
fldlen = dbfldlen ("table.db", 0);
if (fldlen == -1)
  handle the error
```

dbfldlen returns the length of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is returned. An error is likewise returned if there are no open tables, if the file handle is invalid or on some other critical error. If there is an error, -1 is returned and cdberror contains a read-only string describing the error.

19.26 dbflddec

```
int flddec;
...
flddec = dbflddec ("table.db", 0);
if (fldlen == -1)
  handle the error
```

dbflddec returns the length of the field given by fldnum. fldnum is a zero-based offset into an array of names (the first item is 0, for example). If fldnum is less than 0 or greater than the number of fields, an error is returned. An error is likewise returned if there are no open tables, if the file handle is invalid or on some other critical error. If there is an error, -1 is returned and cdberror contains a read-only string describing the error.

19.27 dbcurrent

```
int status;
...
status = dbcurrent ("table.db", "indexname");
if (status == -1)
  handle the error
```

dbcurrent sets the current index for the table to “indexname” and returns 0 on success. It is an error if the table is not open or if “indexname” is not a valid index for the table. If an error occurs, an error is returned and a string describing the error is made available in cdberror.

19.28 dbgo

```
int recno;
...
recno = dbgo ("table.db", recno+1);
if (recno == -1)
  handle the error
```

dbgo sets the current record to the record numbered “recno” (a 32-bit integer value) on success and returns the record number of the current record. It is an error if the table is not open or “recno” is less than one or greater than the number of records. If an error occurs, an error is returned and a string describing the error is returned.

19.29 dbnext

```
int status;
...
status = dbnext ("table.db");
if (status == -1)
  handle the error
if (dbiseof ("table.db"))
  deal with end of table condition
```

dbnext sets the current record in the table to the next physical record without regard for indexing. It is an error if the table is not open. If the current record is already the last record when dbnext is called, dbnext sets the end of file marker on the table to TRUE (which can be checked with a call to dbiseof). If an error occurs, an error is returned and cdberror contains a string describing the error.

19.30 dbnextindex

```
int status;
```

```
...
status = dbnextindex ("table.db");
if (status == -1)
  handle the error
if (dbiseof ("table.db")
  deal with the end of table condition
```

dbnextindex sets the current record in the table to the next record by the current index. It is an error if the table is not open. If there is no current index, dbnextindex decays to a call to dbnext. If the current record is already the last record when dbnextindex is called, dbnextindex sets the end of file marker on the table to TRUE (which can be checked with a call to dbiseof). If an error occurs, an error is returned and cdberror contains a string describing the error.

19.31 dbprev

```
int status;
...
status = dbprev ("table.db");
if (status == -1)
  handle the error
if (dbisbof ("table.db"))
  deal with the end of table condition
```

dbprev sets the current record in the table to the previous physical record. It is an error if the table is not open. If the current record is already the first record when dbprev is called, dbprev sets the beginning of file marker on the table to TRUE (which can be checked with a call to dbisbof). If an error occurs, an error is returned and cdberror contains a string that describes the error.

19.32 dbprevindex

```
int status;
...
status = dbprevindex ("table.db");
if (status == -1)
  handle the error
if (dbisbof ("table.db");
  deal with the end of table condition
```

dbprevindex sets the current record in the table to the previous record by the current index. It is an error if the table is not open. If the current index is not set, dbprevindex decays to a call to dbprev. If the current record is already the first record when dbprevindex is called, dbprevindex sets the beginning of file marker on the table to TRUE (which can be checked with a call to dbisbof).

If an error occurs, an error is returned and cdberror contains a string that describes the error.

19.33 dbhead

```
int status;
...
status = dbhead ("table.db");
if (status == -1)
  handle the error
```

dbhead sets the current record in the table to the first physical record. It is an error if the table is not open. dbhead clears the BOF and EOF markers for the table (sets them to false). if an error occurs, an error (-1) is returned and cdberror contains a string that describes the error.

19.34 dbheadindex

```
int status;
...
status = dbheadindex ("table.db");
if (status == -1)
  handle the error
```

dbheadindex sets the current record in the table to the first record by the current index. It is an error if the table is not open. If there is no current index set, dbheadindex decays to a call to dbhead. dbheadindex clears the BOF and EOF markers for the table (sets them to false). If an error occurs, an error is returned and cdberror contains a string that describes the error.

19.35 dbtail

```
int status;
...
status = dbtail ("table.db");
if (status == -1)
  handle the error
```

dbtail sets the current record in the table to the last physical record. It is an error if the table is not open. dbtail clears the BOF and EOF markers for the table (sets them to false). If an error occurs, an error is returned and cdberror contains a string describing the error.

19.36 dbtailindex

```
int status;
...
status = dbtailindex ("table.db");
if (status == -1)
  handle the error
```

dbtailindex sets the current record in the table to the last record by the current index. It is an error if the table is not open. If there is no current index, dbtailindex decays to a call to dbtail. dbtailindex clears the BOF and EOF markers for the table (sets them to false). If an error occurs, an error is returned and cdberror contains a string that describes the error.

19.37 dbsearchindex

```
int status;
...
status = dbsearchindex ("table.db", "111111111");
if (status == 0 || status == -1)
  handle the error
if (status == 1)
  found it, do something with it
```

dbsearchindex searches using the current index for the value given by "111111111". dbsearchindex does a best fit search and only returns "0" if an error occurs. So, for example, if "111111112" is in the table, the current record pointer would point to that record after this call to dbsearchindex. dbsearchindex clears the end of file and beginning of file conditions. It is an error if the table is not open or it doesn't have a current index set. Also, you will get an error condition if you call dbsearchindex on an empty table. If an error occurs, an error is returned and cdberror contains a string that describes the error.

19.38 dbsearchexact

```
int status;
...
status = dbsearchexact ("table.db", "111111111");
if (status == -1)
  handle the error
if (status == 0)
  didn't find exact match
if (status == 1)
  found it, do something with it
```

dbsearchexact searches using the current index for the value given by "111111111", in this example. This is a bogus Social Security Number. dbsearchexact does

an exact search and returns "0" if the item is not found. It is an error if the table is not open or it doesn't have a current index set. If the item is found, a 1 is returned. If the item is not found or the table is empty, a 0 is returned. If there is an error, -1 is returned. If an error occurs, an error is returned and cdberror contains a string that describes the error.

19.39 dbpack

```
int status;
...
status = dbpack ("table.db");
if (status == -1)
  handle the error
```

dbpack packs the open table given by the handle "table.db" and returns 0 if successful. If an error occurs, an error is returned (-1) and cdberror contains a string describing the error.

19.40 dbreindex

```
#define SIZE 115
int status;
char *cp, table[SIZE+1];
...
status = dbreindex ("table.db");
if (status == -1)
  handle the error
cp = dbopen ("table.db");
if (cp == 0)
  handle the error
strcpy (table, cp);
```

dbreindex reindexes the table given by tablename and returns 0 if successful. It is an error if the table *is* open. So, "table.db" in the dbreindex call is not a table handle. If an error occurs, an error is returned (-1) and cdberror contains a string that describes the error.

19.41 dbexit

```
int status;
...
status = dbexit ();
if (status == -1)
  handle the error
```

dbexit is the only DCDB cint command that does not take any arguments. If there are no open tables, it simply returns 0. If there are open tables, it closes all of them, cleans up the list used to store table handles and returns 0.

19.42 dbtime

dbtime returns a double value that represents the number of seconds since the epoch in fairly high resolution (if your architecture supports it).

19.43 md5sum

```
char *md5;

md5 = md5sum ("test.txt");
if (md5 == 0)
  handle the error
printf ("\nmd5sum of test.txt = %s\n", md5);
```

md5sum will get an md5 hash on the file given by "test.txt". If the user doesn't have the permission to open the file or an error occurs, 0 (NULL) is returned and cdberror contains a string describing the error.

19.44 sha1sum

```
char *sha1;

sha1 = sha1sum ("test.txt");
if (sha1 == 0)
  handle the error
printf ("\nsha1sum of test.txt = %s\n", sha1);
```

sha1sum will get an sha1 hash on the file given by "test.txt". If the user doesn't have the permission to open the file or an error occurs, 0 (NULL) is returned and cdberror contains a string describing the error.

19.45 rmd160sum

```
char *rmd;

rmd = rmd160sum ("test.txt");
if (rmd == 0)
  handle the error
printf ("\nrmd160sum of test.txt = %s\n", rmd);
```

rmd160sum will get an rmd160 hash on the file given by "test.txt". If the user doesn't have the permission to open the file or an error occurs, 0 (NULL) is returned and cdberrror contains a string describing the error.

19.46 dbtime

```
double t1, t2;

t1 = dbtime ();
... do some time consuming stuff here ...
t2 = dbtime ();

printf ("\nIt took %6.4f seconds to do that\n", t2-t1);
```

dbtime will return a double value that is a fairly high resolution timing value (if your architecture supports it). It uses the elapsed() function defined in btime.c.

19.47 dbnumidx

```
int numidx;
...
numidx = dbnumidx ("table.db");
if (numidx == -1)
  handle the error
if (numidx == 0)
  there are no midx's
```

dbnummidx returns the number of multi-field indexes in the table given by tablehandle. It is not an error if there are no multi-field indexes for the table. If an error occurs, dbnummidx returns -1 and cdberror contains a string that describes the error.

19.48 dbismidx

```
int ismulti;
...
ismulti = dbismidx ("table.db");
if (ismulti == -1)
  handle the error
if (ismulti == 0)
  no multi indexes
else
  there are multi indexes
```

dbismidx returns TRUE (1) if there are multi-field indexes associated with the table given by tablehandle table. It returns FALSE (0) otherwise.

19.49 dbmidxname

```
int status;
char *midx;
```

```
...
midx = dbmidxname ("table.db", 0);
if (midx == 0)
  handle the error
status = dbcurrent ("table.db", midx);
if (status == -1)
  handle the error
```

dbmidxname returns the name of the num'th multi-field index for the table given by tablehandle. It is an error if there are no multi-field indexes for the table. Use dbismidx first to determine if there are multi-field indexes for the table. *Note: the result of this call should be used immediately as subsequent calls to certain functions will overwrite it.*

19.50 dbmidxblksz

```
int blksize;
...
blksize = dbmidxblksz ("table.db", "lname_fname");
if (blksize == -1)
  handle error
```

dbmidxblksz returns the blocksize of the multi-field index midxname ("lname_fname") attached to the table given by tablehandle table ("table.db"). It is an error if the table has no multi-field indexes or if it has none called midxname. If an error occurs, -1 is returned and cdberror contains a string that describes the error.

19.51 dbmidxnumfldnames

```
int numfldnms;
...
numfldnms = dbmidxnumfldnames ("table.db", "lname_fname");
if (numfldnms == -1)
  handle error
```

dbmidxnumfldnames returns the number of fields that are used to generate the multi-field index midxname ("lname_fname") which is attached to the table given by tablehandle table ("table.db"). It is an error if the table has no multi-field indexes attached to it or if it has none called midxname. If an error occurs, -1 is returned and cdberror contains a string that describes the error.

19.52 dbmidxfldname

```
char *fldnm;
...
```

```
fldnm = dbmidxfldname ("table.db", "l_fname", 0);
if (0 == fldnm)
  handle the error
```

dbmidxfldname returns the num'th field name that makes up the multi-index midxname ("l_fname") which is attached to the table given by tablehandle "table.db". It is an error if the table has no multi-field indexes attached to it or if it has none named "l_fname" in this case. It is also an error if the multi-field index given by midxname doesn't have num fields (0 in this example). You should call dbmidxnumfldnames to determine how many field names are used in generating the multi-field index. If an error occurs, 0 (NULL) is returned and cdberror contains a string that describes the error. *Note: the result of this call should be used immediately as subsequent calls to certain functions will overwrite it.*

19.53 dbisindexed

```
int isidx;
...
isidx = dbisindexed ("table.db", "SSNumber");
if (isidx == -1)
  handle the error
```

dbisindexed returns TRUE (1) if the field name fieldname in the table given by the tablehandle table is indexed. It returns FALSE (0) otherwise. It is an error if fieldname ("SSNumber") does not exist in the table. If an error occurs, -1 is returned and cdberror contains a string that describes the error.

19.54 dbidxblksz

```
int blksz;
...
blksz = dbidxblksz ("table.db", "SSNumber");
if (blksz == -1)
  handle the error
```

dbidxblksz returns the block size of the index generated for the field "SSNumber" in the table given by the tablehandle "table.db". It is an error if the field is not indexed. You should call dbisindexed to insure that the field is indexed before you call dbidxblksz. If an error occurs, -1 is returned and cdberror contains a string that describes the error.

19.55 dbshowinfo

```
char *tblinfo;
...
```

```
tblinfo = dbshowinfo ("table.db");
if (tblinfo == 0)
  handle the error
printf ("\nThe table info string for table.db is: %s\n", tblinfo);
```

dbshowinfo returns the contents of the info string that is stored in the table header. This is specific to the application and has no meaning as far as DCDB is concerned. It can be a string of up to 125 characters and can contain any value the application programmer desires. It is set with the "info" portion of the "create" command in a .df file. *Note: the result of this call should be used immediately as subsequent calls to certain functions will overwrite it.*

19.56 dbteststring

```
char *tstr;
...
tstr = dbteststring (0, 35);
if (tstr == 0)
  handle error
printf ("%s:", tstr);
```

dbteststring returns a psuedo-random string of characters, all lower-case. It takes 2 arguments, both integers. If the first argument is set, the second must be 0. Conversely, if the second argument is set, the first must be 0. Setting the first argument gives you a string of that length. For example, "dbteststring(10,0)" will return a 10 character string. Setting the second argument will return a string of somewhat random length between 5 and the length specified. So, "dbteststring(0,10)" gives you a random string of lower-case letters between 5 and 10 digits long. It is an error if both arguments are 0 or if both are non-zero. If an error occurs, -1 is returned and cdberror contains a string that describes the error. *Note: the result of this call should be used immediately as subsequent calls to certain functions will overwrite it.*

19.57 dbtestupperstring

```
char *tstr;
...
tstr = dbtestupperstring (0, 64);
if (tstr == 0)
  handle error
printf ("%s:", tstr);
```

dbtestupperstring works the same as dbteststring except that the string returned will consist of upper-case alphabetical digits.

19.58 dbtestmixedstring

```
char *tstr;
...
tstr = dbtestmixedstring (0, 35);
if (tstr == 0)
  handle error
printf ("%s:", tstr);
```

dbtestmixedstring works the same as dbteststring except that the string returned will be of mixed upper and lower-case alphabetical digits.

19.59 dbtestnumber

```
char *tstr;
...
tstr = dbtestnumber (9,0);
if (tstr == 0)
  handle error
printf ("%s:", tstr);
```

dbtestnumber works the same as dbteststring except that it returns a string of numerical digits.

19.60 dbtestnumstring

```
char *tstr;
...
tstr = dbtestnumstring (7,0);
if (tstr == 0)
  handle error
printf ("New license plate number = %s\n", tstr);
```

dbtestnumstring works the same as dbteststring except that it returns a string of combined numerical and upper-case alphabetical digits.

19.61 dbencrypt

```
char *enc;
int size;
...
size = strlen (data);
enc = dbencrypt (data, size, "xOkEj30>");
if (enc == 0)
  handle the error
printf ("\n\nEncrypted data: %s\n", enc);
```

dbencrypt encrypts the data in the data argument using the password in the passwd argument. The size argument should be a multiple of 8, although if it is not, dbencrypt will find the closest multiple of 8 greater than size. The maximum size for a data item is 2048 bytes. The maximum length of the password is 56 bytes (448 bits).

dbencrypt uses the blowfish algorithm. This algorithm is unencumbered with patents and *restrictive* copyrights. It is an open source implementation written by Bruce Schneier (thanks, Bruce). You can use dbencrypt to create and manage passwords or to encrypt data in blocks.

19.62 dbdecrypt

```
char *denc;
...
denc = dbdecrypt (data, size, "xOkEj30>");
if (enc == 0)
  handle the error
printf ("\n\nEncrypted data: %s\n", enc);
```

dbdecrypt decrypts the data given by the data argument using the password given by passwd. The data argument should contain data that was encrypted with dbencrypt and the passwd argument should have the same password used to encrypt the data. The size argument should also be the size of the data before it was encrypted.

19.63 crc32sum

```
char *crcsum;
...
crcsum = crc32sum ("test.txt");
if (crcsum == 0)
  handle the error
printf ("\ncrcsum of test.txt: %s\n", crcsum);
```

crc32sum will generate the crc32 check sum of the file given by fname and return that value. This is the 32 bit check sum code from /usr/src/linux/fs/jffs2/crc32[h,c] from the 2.4.18 kernel modified for my use here.

19.64 bcnumadd

cp = bcnumadd (”24.70”, ”15.17”, 2);

bcnumadd uses the number engine from bc to add the two numbers.

19.65 bcnumsub

total = bcnumsub (”240.37”, ”23.28”, 2);

bcnumsub uses the number engine from bc to subtract the two numbers.

19.66 bcnumcompare

set result [bcnumcompare $num1 $num2 2]
bcnumcompare uses the number engine from bc to compare the two numbers.

19.67 bcnummultiply

total = bcnummultiply ("240.37", "-23.28", 2);
bcnummultiply uses the number engine from bc to multiply the two numbers.

19.68 bcnumdivide

total = bcnumdivide ("240.37", "-23.28", 2);
bcnumdivide uses the number engine from bc to divide the two numbers.

19.69 bcnumraise

total = bcnumraise ("240.37", "-23.28", 2);
bcnumraise uses the number engine from bc to raise the fist number to the power of the second number.

19.70 bcnumiszero

status = bcnumiszero (result);
bcnumiszero uses the GNU bc number engine to determine if arg1 is zero and returns 1 if it is and 0 if it is not.

19.71 bcnumisnearzero

status = bcnumisnearzero (result, 2);
bcnumisnearzero uses the GNU bc number engine to determine if arg1 is near zero. It returns 1 if it is and 0 if it is not.

19.72 bcnumisneg

status = bcnumisneg (result);
bcnumisneg uses the GNU bc number engine to determine if arg1 is negative. It returns 1 if it is and 0 if it is not.

19.73 bcnumuninit

bcnumuninit();
bcnumuninit de-initializes the bcnum stuff (basically, freeing some bcnum numbers that are preallocated). This should be called when you are done with the bcnum bindings. If you don't do it manually, there is an atexit() function that will do it for you. This only returns _OK_.

20 Container Module

The container module is designed to provide a high-performance, generic sorted data structure to scripted languages like Tcl/Tk or Perl. I am sure there are already such beasts for these languages, but as I am unfamiliar with them, I decided to implement one myself. This uses the shell module, a high-performance general purpose sorted module designed for quick adds, quick searches and indexed-sequential retrieval.

The container module is part of the DCDB shared object. If you load the DCDB shared object in a Tcl/Tk or Perl script, you have access to the commands represented here. The documentation here shows examples that are designed for Tcl. If you are using Perl, you need to follow the syntax requirements for Perl.

The container is not very useful in pure C or C++ programs in that you can simply use a shell for these. You loose some performance with the overhead in a container that you would not have with a pure shell, but the performance is good enough overall to make the container worth using with a scripting language.

The basic things you can do with a container are as follows:

- You can create a container with "containerInit".
- You can set field values with the "containerSet" functions.
- You can add a record to a container.
- You can delete a record from a container.
- You can traverse the records in a container, starting at the top and going down or starting at the bottom and going up.
- You can search for a specific item with the "containerSearch" function.
- You can retrieve information from a container record with the "containerGetField" function.
- You can get information about a container and clear errors for a container.
- You can delete a container.

The container module is also provided in the mycint binary when you do a "make cint" in the primary source directory. This builds a special C/C++ interpreter (CINT) that provides DCDB and the container module. You can use the C prototype and the samples provided to get a feel for how to use the container module with CINT. For more information on the CINT bindings, see the primary README file and the README file in the cdb-1.1/src/cint directory.

20.0.1 Container Fields

The container field types are stored in a C enumerated type as follows:

```
typedef enum __contType {
  CTYPE_INT,
  CTYPE_LONG,
  CTYPE_FLOAT,
  CTYPE_DOUBLE,
  CTYPE_STRING,
  /* Add new ones here. */
  CTYPE_UNDEFINED = 0xffffffff
} contType;
```

You can add new ones if you want, although you will need to add the functionality to set values of that type, etc.

Container fields are stored in a header and are accessible via the container handle in the interface functions. There is no other access provided to scripting languages. The container handle is returned by a call to "containerInit" and is used for all other functions in the container module.

20.0.2 Container Error Handling

If an error occurs in a container function, an exception is raised in the scripting language and the error is returned.

There are occasions when an error is not critical (like a unique constraint violation). In that case, you should clear the error condition with "containerClearError" before proceeding to call container functions.

20.1 containerInit

```
char *containerInit (char *fname)
```

containerInit cdf_file_name

The "containerInit" function initializes a container. The argument given by "fname" should be a *container definition file (.cdf)*. The following is an example cdf file:

```
// container example
container "flogcont1"
{
  /* supports C and C++ style comments. */
  "LName" string (64);
  "SSNumber" string (9) unique;
  /* the above allows no duplicates, the below allows dups */
  /* "SSNumber" string (9) dup; */
  "FName" string (35);
  "StreetAddress" string (75);
```

```
  "City" string (45);
  "State" string (2);
  "Zip" string (10);
  "AnnualSalary" float;
  "Age" int;
  "DOB" long;
  /* doubles are also supported...*/
  /* "RealBigAnnualSalary double; */
}
This is ignored...
```

The cdf file supports a C-like syntax. You can use C or C++ comments and whitespace is not relevant. It is an error if you don't use the ';' at the end of each line. Anything after the last close curly brace is ignored.

"containerInit" will return a handle that will be used by all other container routines. This handle should be stored in a variable, as follows:

```
if [ catch { set cont1 [ containerInit container.cdf ] } result ] {
  puts "containerInit container.cdf: $result"
  exit 1
}
```

20.2 containerDelete

```
int containerDelete (char *handle)
```

containerDelete handle

The "containerDelete" function deletes the container given by the handle "handle". On error, it throws an exception and returns an error. Otherwise, it clears the container from memory and the handle "handle" becomes invalid.

20.3 containerClearError

```
int containerClearError (void)
```

containerClearError

The "containerClearError function clears the global error condition for all containers. It takes no arguments and only returns an OK value. Note: the way errors are handled by the container module makes is thread-unsafe.

20.4 containerSetInt

```
int containerSetInt (char *handle, char *field, int value)
```

containerSetInt handle field value

The "containerSetInt" function allows you to set an integer field "field" in the container given by "handle" to "value". It is an error if the handle is not valid or if the field does not exist. Also, if "value" doesn't *look like* an integer to Tcl or Perl, you may get an error from the scripting language. On error, "containerSetInt" throws and exception and returns the error to the scripting language. On success, the field is set to the value. The programmer must set all the fields that she wants populated using the "containerSet" functions before she calls "containerAddRecord".

20.5 containerSetLong

```
int containerSetLong (char *handle, char *field, long value)
```

containerSetLong handle field value

The "containerSetLong" function allows you to set a long integer field "field" in the container given by "handle" to "value". It is an error if the handle is not valid or if the field does not exist. Also, if "value" doesn't *look like* a long integer to Tcl or Perl, you may get an error from the scripting language. On error, "containerSetLong" throws and exception and returns the error to the scripting language. On success, the field is set to the value. The programmer must set all the fields that she wants populated using the "containerSet" functions before she calls "containerAddRecord".

20.6 containerSetFloat

```
int containerSetFloat (char *handle, char *field, float value)
```

containerSetFloat handle field value

The "containerSetFloat" function allows you to set a float field "field" in the container given by "handle" to "value". It is an error if the handle is not valid or if the field does not exist. Also, if "value" doesn't *look like* a float to Tcl or Perl, you may get an error from the scripting language. On error, "containerSetFloat" throws and exception and returns the error to the scripting language. On success, the field is set to the value. The programmer must set all the fields that she wants populated using the "containerSet" functions before she calls "containerAddRecord".

20.7 containerSetDouble

```
int containerSetDouble (char *handle, char *field, double value)
```

containerSetDouble handle field value

The "containerSetDouble" function allows you to set a double field "field" in the container given by "handle" to "value". It is an error if the handle is not

valid or if the field does not exist. Also, if "value" doesn't *look like* a double to Tcl or Perl, you may get an error from the scripting language. On error, "containerSetInt" throws and exception and returns the error to the scripting language. On success, the field is set to the value. The programmer must set all the fields that she wants populated using the "containerSet" functions before she calls "containerAddRecord".

20.8 containerSetString

```
int containerSetString (char *handle, char *field, char *value)
```

containerSetString handle field string_value

The "containerSetString" function sets the field "field" to "value" in the container given by "handle". It is an error if the handle is not valid or if the field does not exist. On error, "containerSetString" throws an exception and returns the error to the scripting language. On success, the field is set to the value up to the max length of the field, if value is longer. In other words, if the string given by "value" is longer than the field length, it will get truncated.

20.9 containerGetField

```
char *containerGetField (char *handle, char *field)
```

containerGetField handle field

containerGetField returns the field value for the field given by "field" for the handle "handle". It is an error if the "handle" or the "field" is invalid. On error, an exception is thrown and the error is returned to the scripting language.

20.10 containerAddRecord

```
int containerAddRecord (char *handle)
```

The "containerAddRecord" function adds the data stored in the container by calls to the "containerSet" functions to the container given by "handle". On success, the record is added to the container and a new internal buffer is allocated and initialized to all zero values. On failure, the record is not added, the buffer remains unchanged, an exception is thrown and the error is returned to the scripting language. Also, if "containerAddRecord" is successful, the current pointer points to the new record in the container.

20.11 containerSearch

```
int containerSearch (char *handle, char *value)
```

The "containerSearch" function searches the container given by "handle" for the value "value" in the sorted field. When you define the container, you specify which field is sorted with the "dup" or "unique" key word. This is the field that is used to search on for "value". If an error occurs, it raises an error and returns the error to the scripting language. Otherwise, it returns false (0) if it doesn't find the value and true (1) if it does. The current container pointer remains unchanged if the "value" isn't found and it points to the found record if "value" is found.

20.12 containerQuery

```
int containerQuery (char *handle, char *value)
```

The "containerQuery" function searches the container given by "handle" for the value "value" in the sorted field. This works similarly to containerSearch() except that it always resets the current pointer unless there's an error. If it can't find an exact match, it will match on the item before the point in the sorted list where the searched for item would be logically added. This only returns 0 (FALSE) on error.

20.13 containerDeleteRecord

```
int containerDeleteRecord (char *handle, char *value)
```

The "containerDeleteRecord" function deletes the current record. The current record can be set by calls to the traversal functions, including "containerSearch", "containerFirst", "containerLast", "containerNext", "containerPrev", etc. On success, the record is deleted from the container. On error, an exception is thrown and an error is returned to the scripting language.

20.14 containerFirst

```
int containerFirst (char *handle)
```

The "containerFirst" function sets the current record in the container given by "handle" to the first record. It's an error if "handle" is an invalid container handle; if so, an exception is thrown and an error returned to the scripting language. Otherwise, the current record is set. It is not an error to call this function on an empty container, but it is meaningless.

20.15 containerLast

```
int containerLast (char *handle)
```

The “containerLast” function sets the current record in the container given by “handle” to the last record. It’s an error if “handle” is an invalid container handle; if so, an exception is thrown and an error returned to the scripting language. Otherwise, the current record is set. It is not an error to call this function on an empty container, but it is meaningless.

20.16 containerNext

```
int containerNext (char *handle)
```

The “containerNext” function sets the current record in the container given by “handle” to the first record. It’s an error if “handle” is an invalid container handle; if so, an exception is thrown and an error returned to the scripting language. Otherwise, the current record is set. It is not an error to call this function on an empty container, but it is meaningless.

20.17 containerPrev

```
int containerPrev (char *handle)
```

The “containerPrev” function sets the current record in the container given by “handle” to the first record. It’s an error if “handle” is an invalid container handle; if so, an exception is thrown and an error returned to the scripting language. Otherwise, the current record is set. It is not an error to call this function on an empty container, but it is meaningless.

20.18 containerBOF

```
int containerBOF (char *handle)
```

The “containerBOF” function returns true if the current record in the container given by “handle” is the first one in the container. If the container is empty or if the current pointer is not on the first record, it returns false. If “handle” is not a valid container handle, “containerBOF” throws an exception and returns an error condition to the scripting language.

20.19 containerEOF

```
int containerEOF (char *handle)
```

The “containerEOF” function returns true if the current record in the container given by “handle” is the last one in the container. If the container is empty or if the current pointer is not on the last record, it returns false. If “handle” is not a valid container handle, “containerEOF” throws an exception and returns an error condition to the scripting language.

20.20 containerNumRecords

```
int containerNumRecords (char *handle)
```

The "containerNumRecords" function returns the number of records stored by the container given by "handle". It is an error if "handle" is an invalid container handle, in which case an exception is thrown and an error is returned to the scripting language. Otherwise, the number of records in the container is returned. The current record pointer remains unchanged by a call to "containerNumRecords".

20.21 containerRestructureIndex

```
int containerRestructureIndex (char *handle)
```

The "containerRestructureIndex" function restructures the in-memory index of the container given by "handle". It is an error if "handle" is invalid, in which case an exception is thrown and an error is returned to the scripting language.

21 DCDB Tcl module

This is the DCDB Tcl module. This is provided in addition to the Tcl/Tk bindings to provide support for high-level functionality that has been shown to be useful in other projects. There is no corresponding perl module.

To use this module, you have to first load cdbtcl.so. Then, you have to append the location of the package to auto_path and then use "package require". The following is an example (ignore the '#' on each line):

```
if [ catch { load ../cdbtcl.so } result ] {
  puts "Couldn't load ../cdbtcl.so: $result"
  return 1
}
lappend auto_path /home/dfmay/devl/cdb-1.1/src/tcl
package require cdb 1.1
```

Of course, this assumes that you have built the pkgIndex.tcl file first. You can use mk_pkg.sh to do that. Also, you are not limited to this location. You can put the cdb.tcl file wherever and append its path to the auto_path variable.

Everything in this package is in the cdb namespace to avoid conflicts with other packages.

21.1 random

proc cdb::randomInit seed

The random functionality consists of several procedures. These are pure TCL procedures that provide psuedo-random functionality. The first procedure is "randomInit". It is called as follows:

```
cdb::randomInit $seed
```

It is not necessary to call this function; the variable "::cdb::seed" is set to the value of [clock clicks -milliseconds] which should be sufficient for most purposes.

proc cdb::random

The next procedure in random group is called "random". This function simply calculates a random value between 0 and 1 based on the seed that was either provided or calculated from the current time. It is used as follows:

```
set randValue [ cdb::random ]
```

If you use the random procedure and want to generate another random value, you should reseed with a call to "cdb::randomInit".

proc cdb::randomRange range

The "randomRange" procedure is like the random procedure except that it returns a psuedo-random integer between 0 and the "range" value provided as an argument. It is used as follows:

```
set seed [ clock clicks -milliseconds ]
cdb::randomInit $seed
set randVal [ cdb::randomRange 100 ]
...use $randVal
```

21.2 isValidTable

proc cdb::isValidTable table

The "isValidTable" procedure returns 0 (false) if the "table" argument is either not a valid table handle or is a table handle but the table pointed to by it is not open. If the "table" argument is valid and points to an open table, "isValidTable" returns 1 (true).

21.3 listTable

proc cdb::listTable table

The "listTable" procedure takes as an argument a valid table handle given by "table". If successful, "listTable" populates a list called "::cdb::tableListing" that is part of the cdb namespace with formatted data about the list, including number of records, information about the fields in the table and information about the indexes that are part of the table. Then, listTable returns 0.

The output of "listTable" is pre-formatted by the function in a way that is deemed to be attractive. If the caller wants to format the output in another way, the data in the list will need to be extracted first. There is no provision for altering the format of how data is presented.

If the "table" argument is not a valid table handle or there is an error, the "::cdb::tableListing" list will have as a first value a string that indicates what the error was and "listTable" will return 1.

Use the listTable procedure as follows:

```
set status [ cdb::listTable $table ]
if { $status } {
  ... handle the error.
} else {
  set len [ llength $cdb::tableListing ]
  for { set i 0 } { $i < $len } { incr i 1 } {
    set outstr [ lindex $cdb::tableListing $i ]
    puts $outstr
  }
}
```

21.4 listTableArray

The "listTableArray" procedure is similar to the "listTable" procedure in that it provides information about the table given as the argument "table". However, "listTableArray doesn't provide as much information. Also, the information is not formatted. The information is returned in an array called "::cdb::tableArray". The information contained in the "cdb::tableArray" array is as follows:

```
tableArray(error) - only set if there is an error
tableArray(numfields) - number of fields in the table
tableArray(0,name) - field name of field 0
tableArray(0,indexed) - true if indexed, false otherwise
tableArray(0,fieldtype) - type of field 0
tableArray(0,fieldlen) - length of field 0
tableArray(0,fielddec) - length of decimal portion of field 0 (if applicable)
...
tableArray($i,name) - field name for $i
tableArray($i,indexed) - true if indexed, false otherwise
tableArray($i,fieldtype) - type of field $i
tableArray($i,fieldlen) - length of field $i
tableArray($i,fielddec) - length of decimal portion of field $i (if applicable)
```

The $i value ranges from 0 to numfields-1, so for each field, there are 5 entries (name, indexed, fieldtype, fieldlen, and fielddec).

You would use the "listTableArray" procedure as follows:

```
set status [ cdb::listTableArray $table ]
if { $status } {
  puts -nonewline "***Error on tableArray $table: $cdb::tableArray(error)"
} else {
  set outstr [ format "There are %d fields in $table" $cdb::tableArray(numfields) ]
  puts $outstr
  puts ""
  for { set i 0 } { $i < $cdb::tableArray(numfields) } { incr i 1 } {
    set outstr [ format "Field %s, type %s, len %d, declen %d, indexed %s" \
      $cdb::tableArray($i,name) \
      $cdb::tableArray($i,fieldtype) \
```

```
      $cdb::tableArray($i,fieldlen) \
      $cdb::tableArray($i,fielddec) \
      $cdb::tableArray($i,indexed) ]
    puts $outstr
  }
  puts ""
}
```

21.5 fileModeString

cdb::fileModeString fname

The "fileModeString" procedure takes the name of a file as an argument and returns a mode string similar to the mode string shown by 'ls -l', something like '-rw-r–r–' for a normal file with 644 permissions. The difference between 'ls -l' and "fileModeString" has to do with how "fileModeString" handles symbolic links. "fileModeString" will return the mode string for the file linked to, whereas 'ls -l' returns lrwxrwxrwx. I felt the mode string for the file being pointed to is more useful than the other.

The file mode is returned in the variable "::cdb::fileMode", which is in the cdb namespace. If an error occurs, "fileModeString" returns a non-zero value and "cdb::fileMode" contains the error. If there is no error, the return value will be 0 and "cdb::fileMode" will contain the mode string. "fileModeString" would be used as follows:

```
set status [ cdb::fileModeString /bin/ls ]
if { $status } {
  puts "***Error calling cdb::fileModeString: $cdb::fileMode"
} else {
  puts "Mode of /bin/ls: $cdb::fileMode"
}
```

Index