MF

1.03

Severe-performance database for professional applications

Screwy looking fonts:

If your fonts seem all out of wack, try changing the selected printer. When preparing this manual, it was created for an HP DeskJet 500 or a Laserjet. If you are using something other than one of these, your fonts may look strange. Sorry about this, but it wasn't noticed until we tried to FAX it and the fonts got all out of wack...

Overview

WHAT IT IS

MF is similar to many DBMS's available today, however, it is generally faster and smaller than the others. Using standard functions you will get a speed increase of about 100% over a lot of other DBMSs. However, using the severe-performance functions, speed can be UP TO 20,000% faster. (Note: we have not benchmarked ourselves against all other databases, so it is impossible to say there isn't something FASTER. But, we are confident that we are pretty quick...)

MF also allows professional developers to create 'CLIENT-SERVER' based programs without requiring the developer to pay heavy run-time license fees for every customer. (NOTE: BIOS drivers are not currently available...) The 'severe-performance' functions will permit C/S type of access across a network. (Since record based implementations don't go well with the C/S architecture...)

MF is created for software-developers with a need for speed and total flexibility. You can treat MF as the worlds largest linked-list or as a completely relational data-storage system. MF is for programmers that want to get very low level. Please, don't try to use MF if you're comparing it to Microsoft Access or some other high-level database. MF is designed to be embedded in professional applications -- not used for a quick inhouse report generator.

You will probably discover a bunch of neat things about MF that you couldn't do in your old DB. What you won't discover about MF is a limitation! If you find a limitation that is correctable in the DOS world, we will correct it. (e.g. We can't create a 10 Terabyte data file without circumventing DOS. So, we can not correct this terrible <grin> limitation.) However, future releases will support multiple database splitting to allow 2 BILLION records (unlimited total disk space). And, if a few compilers get their act together and support a 64-bit long, then we'll support gillions of records...

WHAT IT IS NOT

Where MF is not as complete (in screen I/O, miscellaneous overhead), is where other DBMS' outshine MF. But, if you are using a high-level interface tool, MF will fit right in. MF is not tied to a particular language. MF will work with VB and C and should work with any language capable of calling a DLL.

MF does not support ODBC, SQL, etc... MF is the low-level version of a high-level database. You may create an SQL wrapper for MF and sell that with MF (we don't charge a redistribution fee -- so, you can feel free to add to it whatever you want...). If you create an ODBC driver or whatever, it's all yours.

Also, if you create an 'extension' to MF, you may distribute MF at no-charge (provided, the end-user of your extension pays something to you and us...) Of course, you could pay for the distribution rights and the end-user wouldn't have to pay us one cent. (But, you'd be left to support the actual database itself...)

Why choose MF?

MF was created to satisfy a need for a windows database that actually worked without GPF'ing or behaving inconsistently every couple of hours. MF is actually owned by a parent company, but I (Carl Brown) wrote it and convinced them to let me put it out for public use.

MF is tested extensively on in-house (actually, semi-retail...) products where I work. Before MF is

released, it is placed in to the production release of 3 different retail applications and most bugs are discovered before the ZIP file (you receive) goes out the door. Chances are, there aren't any bugs in the version you receive. However, annomalies do pop up from time to time and I appoligize for any inconveniences that occur.

So, why choose MF?
It's cheap, it's fast, it's embedable, it's small, it's expandable, and it is actually tested in a real world application before it goes out the door.

Concepts

The section assumes you are familiar with at least one other database manager and are an experienced programmer. With a few exceptions, most examples will be in VB. Reason: it is easier for a C programmer to understand VB than a VB programmer to understand C.

DATA FILES

MF does not hold you to a set of fields. MF merely allocates storage to the size you specify. A data file consists of:
INDEX_KEYS and DATA_SPACE.

DATA_SPACE can be up to (32k minus the size of the INDEX_KEYS). Unless you plan to store BLOBS, that should be more than enough space...(BLOB support will be added in future releases. How far in the future depends on how many people request it...)

A record in MF is 'virtual', meaning in can be a record of any type for you.
e.g.
in C, you will often find structures of 'unions'. These unions can mean different things at different times.

Therefore, in MF, you just tell it the size required for the largest 'structure' you will use. More often than not, you will only be dealing with one structure per file, but in some cases, you may not. MF leaves the decision up to you...

e.g.
C
```
typedef struct {
      char lName[20];
      char fName[20];
} tPersonKey;

typedef struct {
      char SAddr[100];
      char Zip[9];
      char City[20];
      char SSN[9];
} tPersonData;

typedef struct {
      tPersonKey Key;
      tPersonData Data;
} tPerson;
```

To create a database like this you would give the database record size:
```
      sizeof(tPersonData);
```
and the first index key a size of:
```
      sizeof(tPersonKey);
```

How you go about setting it all up is entirely up to you... But, once you have your structures defined, you can manipulate and massage them and 'FILL' them however you want to...

NOTE: MF is not sensitive to the NULL terminator. It is recommended that you NULL out (or come up with some consistent way) of identifying your data. (Especially concatenated index fields...)

VB Example

```
Type tPersonKey
      lName as string * 20
      fName as string * 20
End Type
Type tPersonData
      SAddr as string * 20
      Zip as string * 9
      City as string * 20
      SSN as string * 9
End Type

Type tPerson
      Key as tPersonKey
      Data as tPersonData
End Type
```

Size of data would be:

```
      len(tPersonData)
```

Size of key would be:

```
      len(tPersonKey)
```

(NOTE: To get the LEN in VB, you must first declare those structures (Types) as a variable and get the len of the variable.)

INDEXES

Indexes can be keyed on:
character
(Char fixed length up to 128 bytes)
characters are based on ASCII ordering. e.g. A > a
(NOT sensitive to NULL)

character - NOCASE
(Char fixed length up to 128 bytes)
strings will not be case sensitive. e.g. A == a
(NOT sensitive to NULL)

integer - signed
(2 Bytes)
(Arrays of integers are allowed up to 128 bytes)

long - signed
(4 Bytes)
(Arrays of longs are allowed up to 128 bytes)

User-Defined
(Up to 128 byte keys)
(Anything you want...)

Combination keys are not supported (with correct results) internally (i.e. String and Integer combination). That's what user-defined is for... See *Creating User-Defined keys* for more information.

For the sake of argument, a 'key' that has an 'integer' tacked onto the end will sort, but it won't sort correctly. 'Integer's are stored with the LSB (least significant byte) first. Which means 255 > 256. While you could 'SWAB' the values and get correct results with concatenated string and integer combos, it's probably more trouble than it's worth.

Internally, you may concatenate similar data types (i.e. longs, strings and ints). e.g.
VB

```
TYPE Key1
      Group as Integer
      SubGroup as Integer
END TYPE
```

C

```
typedef struct {
      int   Group;
      int   SubGroup;
} Key1;
```

With this example, when you create the index, you would specify a type of INTEGER and a byte count of 4. To MF this will imply: Order by the First and sub-order by the second integer. This is probably one of the most powerful capabilities of MF. Many databases require you to turn these into STRINGS and concatenate the strings for your search/storage. You will be surprised at how fast this is and how useful it is. (especially when dealing with longs...)

As an example, 3 LONGS would require only 12 bytes in the index (in MF). However, in many other databases, to store the string equivalent would be over 30 bytes. Additionally, if you stored the

'longs' as a string, you would have to PAD the string with 0's in order to get the database to store them in 'ascending' order.

NOTE:
EACH index must be the first part of the 'structure'. e.g.

```
TYPE tRecord
      ' Index 0
      SSN as string * 11

      ' Index 1
      LastName as string * 20
      FirstName as string * 15

      ' DATA (unindexed portion of the record...)
      Street as string * 50
      City as string * 20
      .
      .
      .
End Type
```

Why did we do it this way? Because, if you're NOT updating an index field, we can stream a contiguous data segment to disk and that's MUCH, MUCH faster than scattered writes.

There is one problem with this approach: What if you want to use a key in two different indexes? Unfortunately, the only solution is to store it in the database twice (or more, depending on how many indexes need the key...). Certainly this is a nuisance, however, the extra use of space shouldn't be too burdensome.

Another note about this: This 'structure' has the keys and data defined in the same structure. That is perfectly acceptable. The only problem with defining the indexes and data in the same structure is if you want to write the 'data-only' portion of a record. In C, you could pass the first 'data' member of the structure and pretend it is the data structure. In VB, it isn't quite so easy to segregate the two. Your better off if you have separate index and data structures because if you add more data fields and you want a newly added data field to be the FIRST data field, all of your 'member' passing would need to be fixed to refer to the first member.

Putting it together

(Quick start)

If you have VB, please read the sample VB application. It demonstrates how you would go about developing an application using MF. This is a quick review of how to use MF.

From begining to end:

```
mfInit
mfOpen and mfCreate

mfRead/Write/Seek/etc

mfDeInit

Browse through Appendix D (Quick Reference)
```

An explanation of begining to end:

`mfInit`

In your WinMain or startup form, you would generally call mfInit. The value returned from mfInit is used in nearly every call to MF. You should only call this one time (at the beginning of your program). In general, the value returned from this call should be placed in a GLOBAL variable so you don't have to reinit every time you need your handle. If you are opposed to GLOBALs, you might think about a static wrapper that will return your MF TASK handle (or, will INIT if it doesn't have one yet...).

Recommended use (in C):

```
hMF_TASK    hMFTASK;
WinMain(...)
{
.
.
.
hMFTASK = mfInit(...);
.
.
.
}
```

`mfOpen and maybe mfCreate`

Most the time, you will need access to your databases immediately. Generally, just open them and keep them open (Or, create and open if they don't exist...). (Note: You may wish to make your database handles 'GLOBALs' also. You should only open a particular database ONE time per application instance or mfClose call...).

`mfRead/Write/Seek/Skip`

You will probably repeatedly call the Read/Write/Seek and Skip functions (as well as a few others...). These 'other' functions are generally used in response to user actions (Dialog box response, listbox clicks, etc...).

Nearly every MF function relies on the handles supplied by mfInit and mfOpen.

mfDeInit and maybe mfClose

When your application is about to exit, make sure you call the mfDeInit function. You MAY close your databases yourself (mfClose) or you may allow mfDeInit to do it for you. Generally, you place the mfDeInit in your WM_CLOSE message or in the Form_Unload of your main form (in VB).

Document Conventions

Most C programmers will recognize everything in here, however, to be fare to VB programmers, the following will apply:

&	in front of a 'variable' means: Pass by reference
i	in front of a variable means signed integer
l	in front of a variable means signed long
s	in front of a variable means array of characters (a string...)
sz	in front of a variable means array of characters (a string...) with NULL termination
a	in front of a variable means 'any' type of data

Standard API Calls

NOTE: All API calls return a NEGATIVE if there is an error in the process. No API call will return a negative unless it is an error. (EOF and BOF are the only exceptions...). Therefore, if the return value is NOT negative, you can assume the function was successful.

mfAppendData

Appends a record to a database -- returns new record #.

```
lNewRecordNumber = mfAppendData(&sDataOnly, iTask, iDBHndl)
```

parameters:

- Data	DATA ONLY portion of a record structure
- Task	(hMF_TASK) Task from mfInit
- Database	(hMF_DB) Database handle from mfOpen

returns: (RPTR) New Record Number or error (-x)

Append will place only the 'DATA' portion of a record on disk. The 'indexes' are not updated until you do a mfWrite. Generally, you will call:

```
mfWrite(mfAppendData(...), ...)
```

mfAppendData should probably be called mfNewRecord or something else. An append implies the record is placed at the end of the database. In most cases, it is. However, if there are any deleted records, mfAppendData will reuse a deleted record.

NOTES:
There is a reason for passing the DATA and only writing it (and not automatically updating the index keys while we are at it...). As mentioned in mfWrite, the slowest part of accessing the database is the index. So, in theory, you COULD append 200 records and pass the 'indexing' part off to a server program that you create. This is just an idea that we will fully integrate into the system in future releases... However, you can take advantage of the enhanced speed (if you wish) right now. Don't fret if you don't. We usually just append and write at the same time. It's only there so you have the option.

mfBottom

Returns bottom (last) record in index order.

```
lBottomRecord = mfBottom(iTASK, iDBHndl, iXHndl)
```

parameters:
- Task (hMF_TASK) Task from mfInit
- Database (hMF_DB) Database handle from mfOpen
- Index Number (int) Index that you want the 'last' record from

returns: (RPTR) last record in index order or error (-x)

If you want to start at the bottom of an index and read through all the records in the database in 'index' order, you can call this to get the last record in the index, then call mfSkip to move sequentially through the database.

Note that BOTTOM is not the bottom of the database, but the bottom of the index you specify. (i.e. the LAST record in sorted order). To obtain the last record in the database, see mfInfoDB.

mfBottom is generally used to return to the last record if the user hits EOF or if you need to find the last 'transaction' or what-not that was added to the database.

Code Fragment (VB):

```
' Demonstrates a 'skip' routine (a more interesting skip routine is in
' the BCARD.frm file)
' If the SKIP fails to produce a valid record, back-up to the 'LAST'
' record in the database

lCurRec = mfSkip(lCurRec, 1, iTASK, iDBHndl, iXHndl)
      ' If the user is past the EOF, beep at them and
      ' place them on the last record.
if lCurRec = MFSEEK_EOF then
      BEEP
      lCurRec = mfBottom(iTASK, iDBHndl, iXHndl)
endif

mfRead(lCurRec,...)
.
.
. ' Update the display...
```

mfClose

Closes a database and all associated indexes

```
iStatus = mfClose(iTask, iDBHndl)
```

parameters:
- Task (hMF_TASK) Task from mfInit
- DB Handle (hMF_DB) Handle returned from mfOpen

returns: 0 or error (-x)

When you are done with a database, you can close it with this command.

MF can open and close databases VERY quickly (as fast as DOS will allow...). Since there are a limited number of file-handles available, we recommend closing any database you will not be using often.

Note: mfDeInit will automatically close all open databases and indexes for you.

mfCreateDB

Create a new database and indexes

This is the most convoluted of all the commands in the MF. But, lets give it a shot:

```
iStatus = mfCreateDB(&sFileName, iDBRecSize, iNumIndexes,
                        &iRecSizeArray, &iTypeArray)
```

parameters:

- Filename — Path and Name of database to create (8 character name only)
- Record Size — Size of the 'DATA-ONLY' field to create (up to MF_MAXREC_SIZE)
- # of index's — # of index's for this data file (up to MF_MAX_NUM_INDEX)
- Size of each — This is an array of integers containing the size of each index you will use in the index (ordinal) (up to MF_MAX_KEY_SIZE)
- Type of each — array of integers, again, that tells MF the TYPE of each index (MFCOMP_CHAR, UDK value, MFCOMP_INT, etc...) This has a one-to-one correlation with the 'Size of each' parameter

returns: 0 - OK, -x error

Code Fragment (VB):

```
Dim file$, recsize%
Dim person As tPerson         ' Key 0
Dim company As tCompany       ' Key 1
Dim ref As tref               ' Key 2
Dim bCard As tCard                  ' KEYS and Data combined to 1 structure

file = "C:\DATA\MYDB"

' Calculate the size of an individual records data
recsize = Len(bCard) - Len(person) - Len(company) - Len(ref)

' Fill arrays with index parameters (tintArray defined in mf.BAS)
ReDim indSize(0 To 2) As tintArray  ' There will be 3 indexes total
ReDim indType(0 To 2) As tintArray  ' Array for the index 'TYPES'

      ' Set the SIZE of each key so mf can allocate the space
      ' for it
indSize(0).i = Len(person)  ' Key 0
indSize(1).i = Len(company) ' Key 1
indSize(2).i = Len(ref)     ' Key 2

      ' The type of each index
indType(0).i = MFCOMP_CHAR  ' CHAR key - case sensitive
indType(1).i = 1001         ' UDK - Sorts in 'reverse' order...
                          ' (see mfUDK.c for example)
indType(2).i = MFCOMP_INT   ' An integer key...

If mfCreateDB(file, recsize, 3, indSize(0), indType(0)) > -1 Then
      MsgBox "File Created Successfully"
Else
      MsgBox "Error creating database"
End If
```

Code Fragement (C):

```
/*
      This (the typedefs) would probably be in a .h file somewhere, but
      for demonstration purposes, it's here...
*/
typedef struct {
      char  szLName[20];
      char  szFName[20];
} tPersonName;

typedef struct {
      int   iAge;
} tPersonAge;

typedef struct {
      char  szStreet[20];
      char  szCity[20];
} tPersonData;

typedef struct {
      tPersonName tName;
      tPersonAge  tAge;
      tPersonData tData;
}tPersonRecord;

int   iaSize[2];  // We have 2 indexes, so we need 2 positions
int   iaType[2];  // Ditto...

iaSize[0] = sizeof(tPersonName);
iaType[0] = MFCOMP_CHARIC;

iaSize[1] = sizeof(tPersonAge);
iaType[1] = MFCOMP_INT;


if (mfCreateDB(   "SomeDB",                           // Name of DB
                  sizeof(tPersonData),                // Size of DATA
                  2,                                  // # of indexes
                  iaSize,                             // size of each index
                  iaType)                             // Each index 'type'
                  > -1)
      MessageBox(NULL, "Successful", "NOTE", MB_OK);
else
      MessageBox(NULL, "Failed", "NOTE", MB_OK);
```

WARNING:
The MINIMUM size for a record is 4 BYTES. The (total) size of your indexes apply towards the minimum.

NOTES:
- Filenames/Paths should NOT have a .xxx extension. Pass ONLY the path/filename that you want.

e.g.

VALID:	C:\DATA\PEOPLE
INVALID:	C:\DATA\PEOPLE.DB

- mfCreate does NOT open the database, it only creates it. You must use an mfOpen to open it. Do not try to Create an Open (open by you or anyone else...) database. This works (for some ungodly reason!) and will corrupt any database handles (and usually GPF) the system...

- MF will create a database with the name you passed and will create a 'database.nnn' extension for each index.

e.g.

C:\DATA\PEOPLE	- You pass this...

MF will create:

C:\DATA\PEOPLE	- Database
C:\DATA\PEOPLE.0	- Index for first index in list of indexes array
C:\DATA\PEOPLE.1	- Index for second index in list of indexes array

.
,
.

mfCreateIndex

Create index will re-create a 'missing' or otherwise corrupted index

iStatus = mfCreateIndex(szFileName, iIndexRecSize, iDataType, iIndexNumber)

parameters:
- the path and 'name' of the database itself (no extension)
- the 'size' of the index field (2=int, 4=long, x = char, etc...)
- the DATA type (MFCOMP_LONG, etc...)
- the 'index' # it is. (when you first created your array of indexes to be created, you started (errrr, MF started) at zero. So, if you needed to recreate the first index (# zero), you would pass a zero.)

e.g. (*C*)
```
int iISizeArray[3];
int iITypeArray[3];

iISizeArray[0]= 4
iITypeArray[0] = MFCOMP_LONG;
.
.
.
mfCreateDB(...);
```

At a later date, if you needed to recreate that FIRST index, you could do a:
```
mfCreateIndex("C:/PATH/SameDB", 4, MFCOMP_LONG, 0);
```

mfCreateIndex is automatically called by the mfCreateDB function. Unfortunately, you cannot ADD indexes after the initial database creation (without re-creating the database). But, you could re-build an identical index from scratch.

The new index MUST have the same SIZE (parameter 2) as the old index. However, you CAN change the type of the index without causing a problem. When you call mfCreateIndex, the old index is completely destroyed. You could use mfReIndex to repopulate the index.

NOTE:
This is not a recommended function. There are very few reasons to use this function. The only possible reason is if you lose a sector on your hard drive -- and that sector happened to contain the first 100 bytes (the header information) of the index... If you are considering this function, consider an alternative. A better approach would probably be to create a new database and append all the data in the current database in to the new database.

mfDeInit

Tells MF to deallocate any space it reserved for your task and to close all open databases and indexes you opened (in your task...). (Also, releases the TASK so other applications can use it and unloads and extensions loaded for this task.)

```
iStatus = mfDeInit(iTask)
```

parameters:
TASK (hMF_TASK) Task number returned by mfInit

returns: 0 - OK, -x error.

You should always call mfDeInit when you are through with MF. If your application is the only application using MF, then it is rather irrelevant if you call mfDeInit. (Windows will toss MF out of memory when the last application using MF ends).

A nice feature of mfDeInit is to close all your DB handles. If you have 'lost track' of the open DB handles in your application, mfDeInit will close everything you have open. As to why you lost track -- that may be of some concern. However, if you aren't sure, you can always use it. A potential time is with the use of the mfReIndex command. Since reindexing may cause your DB handles to change, it may be prudent (or easier) to deinit and reopen what you need.

mfDelete

Deletes a record.

```
iStatus = mfDelete(lRecordNumberToDelete, iTask, iDBHndl)
```

parameters:
- Record Number (RPTR) record number you want removed
- Task (hMF_TASK) Task from mfInit
- Database (hMF_DB) Database handle from mfOpen

returns: 0 - all OK, or -x Error

see also
mfIsDeleted

This is a 'false' delete. Or, maybe it is a 'true' delete. We are not sure which. However, when a record is deleted, it really is deleted. The database will automatically re-use the deleted record on the next 'mfAppendData'. In either case, ALL data in the record is set to NULL and it is removed from all indexes (It CAN NOT be found during a SKIP or SEEK...).

It is POSSIBLE to access the record by SPECIFICALLY referencing it, however, there is no real reason to do so. Rarely would you want to skip through a database by ACTUAL record position. DO NOT read/write a deleted record. If you write to it, you will destroy the linked-list of deleted records. If you READ it, you will get garbage.

mfInfoDB

(release 1.02 - enhanced)
Sets passed parameters to the status of the database.

iStatus = mfInfoDB(&iRecSize, &iNumIndexs, &lNumRecs, &lNumLiveRecs, iTask, iDBHndl)

parameters:
- Record Size 'Size' of record will be set in this variable
- # of index's Total # of index's defined for this DB
- # of Records Number of records in this DB
- # live recs Number of live (not deleted) records in the database (new to 1.02)
- TASK Task handle given to you by mfInit
- DB Handle Database handle given to you by mfOpen

returns: 0 - OK or error (-x)

TIPS: You may set any of the first 4 parameters to NULL if you do not wish to retrieve that value. This alleviates you from declaring variables that you don't need.

mfInfoIndex

Returns # of bytes in index key

iBytesInKey = mfInfoDB(iTask, iDBHndl, iNumberOfIndex)

parameters:
- TASK Task handle given to you by mfInit
- DB Handle Database handle given to you by mfOpen
- Index # Index # you want the # of bytes in the KEY for

returns: # of bytes in key (this is the value used when the index was created)
-x - Error Code

mfInit

Tells MF to allocate space for you (internally) and gives you a TASK ID.

```
iTask = mfInit(&extensionDLLstructure )
```

parameters:
- Extension DLL pointer to extDLL structure

returns: The magical 'Task' number or error (-x)

see also:
mfDeInit

This function returns the magical 'task' # that is used in every function in the system. You should only call this 1 time PER APPLICATION INSTANCE. The task handle this function returns is used throughout your program. When your program exits, you should call the `mfDeInit` function.

A maximum of 10 TASKS can be active at one time. This is, usually, not a problem since MS-Windows wont usually support more than about 10 programs running at once...(and, unless MF is ridiculously popular, it is highly unlikely, all 10 will require access to MF <g>).

However, if for some STRANGE reason, this is a problem for you, let us know. We'll fix it (and we probably will, anyway, in a future release...). Also, as a temporary measure, you could rename MF.DLL to something else and load the other name. The net-effect is that you'll get 10 more TASK handles...

EXTENSION DLL's
The only (published and debugged) extension currently supported is that of User-Defined keys. See *Creating User-Defined Keys* for more information regarding creating extensions to MF.

However, we must still talk about this parameter (whether you choose to use it or not).

```
' This is defined in MF.BAS (and MF.H for C)
Type tExtDLL
   type As Integer                  ' Type of extension
   DLLName As String * 128          ' 'FILENAME' of DLL for extension
End Type

' If you DO NOT use an extension:
   ReDim extDLLs(0 To 0) As tExtDLL
   extDLLs(0).type = -1       ' tells MF there are no more extensions

' If you DO use an extension:
   ReDim extDLLs(0 To 1) As tExtDLL
   extDLLs(0).type = MFCOMP_UDK    ' tells mf to use this dll for UDK's
   extDLLs(0).DLLName = "mfUDK.dll"
   extDLLs(1).type = -1    ' tells MF there are no more extensions

' In either case, you must call the mfInit function
   TaskHndl = mfInit(extDLLs(0))
```

MF will automatically load the DLL containing any extensions and use them for THIS task only. (Other tasks may use a different set of extensions...)

mfIsDeleted

(release 1.02)

Checks if a record has been deleted (by mfDelete)

iDeleted = mfIsDeleted(lRecordNumber, iTask, iDBHndl)

parameters:

```
- Record    Record # to check for delete status
- TASK            Your task number
- DB        The database handle
```

returns:

```
0 - Not Deleted, 1 - Deleted.  (-x is an error...)
```

Normally, you should not need to check the 'delete' status of a record. However, if you must process records in 'record' order, you may need to know if the record has been deleted. A record will never show up in an INDEX if it has been deleted. (i.e. mfSkip/mfSeek will never return a record number that has been deleted...)

mfLock
(release 1.02)

Locks a record

iStatus = mfLock(lRecordNumber, iTask, iDBHndl)

parameters:
```
- Record    Record # to lock
- TASK            Your task number
- DB        The database handle the record is in
```

returns:
```
0  - got the lock,
-1 - File already locked
```

see also:
```
mfUnLock
```

You must maintain the lock and unlock status of all records.
If you lock a record -- be sure to UNLOCK a record. If you do not, that record will remain locked until:
- You close all the databases
- The user disconnects from the network
- The user quits windows

You may have as many records locked as your operating system will allow. Generally speaking, you should only lock one record at a time. The typical time to do so is if a user requests to edit the data in a particular record. You would lock the record until the user finished their changes and then write the record and unlock it for others to use.

In order to use this effectively, you must make a concious decision to lock and unlock records. In this release, MF will allow another user to write to a LOCKED record. It is up to you to decide when a record can and can not be written to. mfLock will tell you that a record is already locked -- so, you can make the choice to NOT allow that user to have exclusive rights to a record.

Typically:

- A user selects to 'EDIT' a record.
- You attempt to LOCK the record.
- If the record won't lock -- you tell the user that another user has the record locked and tell them to go pound sand.
- Otherwise, you allow the user to edit. When the user selects to SAVE (or CANCEL) you would mfUnLock the record.

mfOpen

Opens a database and all associated indexes

```
iDBHndl = mfOpen(&szDatabaseFileToOpen, iTASK)
```

parameters:
- Filename (ASCIIZ string) Path and Filename of database to open
- TASK Task handle given to you by mfInit

returns: (hMF_DB) Handle to database (used in nearly every call to MF)
-x - Error Code

See also: *mfClose, mfDeInit*

A database needs to be opened only once during program execution. A MAXIMUM of 14 databases may be in use per TASK. (Up to 9 indexes are automatically opened when you open their associated database. The index's are 0 through 8 and correlate to the 'order' in which you defined them in the `mfCreateDB`.)

NOTES:
A common misconception is to register for a new task (mfInit) each time you need a new database (mfOpen). However, a single task can handle 14 database (and their associated indexes). Most applications should only call mfInit one (1) time throughout the application execution.

The return value of this function is used in most mfXxx calls.

mfRead

Reads an existing record.

```
iStatus = mfRead (lRecordToRead, &sData, iTASK, iDBHndl, iOption )
```

parameters:

- Record # to read	Record # in the database that you want to load into the data buffer
- Data buffer	a buffer (YOU created using a structure/'type') to load into
- Task	Task returned from mfInit
- DB	DB returned from mfOpen
- Option	MFRW_ALL, MFRW_DATA, MFRW_KEY

returns: 0 - All OK, or error (-x)

Options explained:

MFRW_ALL specifies to read the KEY and the DATA into the buffer
MFRW_DATA will load only the DATA portion of the record
MFRW_KEY will load only the KEY portion of the record

Note: There is minimal performance difference for these options in the mfRead call. However, there are major performance differences in the mfWrite call.

Normally, you will use MFRW_ALL. The other options are primarily for tuning performance.
e.g. (using the structures defined at the beginning of this guide)

Code Fragments:

' This will load ONLY the 'key' for this record into the 'key' structure
```
dim keyOnly as tPersonKey
junk = mfRead(1, keyOnly, TASK, DB, MFRW_KEY)
```

' This will load only the DATA for this record (no key information)
```
dim dataOnly as tPersonData
junk = mfRead(1, dataOnly, TASK, DB, MFRW_DATA)
```

' This will load all the data for this record
```
dim aPerson as tPerson
junk = mfRead(1, aPerson, TASK, DB, MFRW_ALL)
```

mfReIndex

(release 1.02)

Reindexes a database

```
iDBHndl = mfReIndex (HwndStatusDisplay, iTASK, iDBHndl)
```

parameters:

- Status Display	Since Reindexing can be a rather lengthy process, you can have it send a 'status' update to a window. The passed 'window' must respond to the WM_SETTEXT (from SendMessage) in order to see the update. (This parameter may be NULL if you don't want an update).
- Task	Task returned from mfInit
- DB	DB returned from mfOpen

Reindex will destroy the existing indexes and recreate them from the original data. If you think a database index may be corrupted, this will (hopefully) fix it for you.

Reindex may take a very long time if there are a lot of records. Hence, you can pass it a handle to a window that will recieve a 'percent complete' update. If you wish to have a status bar, you can take this number and convert it to any status bar you wish (or, have access to...). The percent complete will be a single string that can be converted to an integer with little difficulty. (see the sample application for some ideas of processing the WM_SETTEXT update...). mfReIndex will send a message at each percent complete, e.g.

0
1
2
3
.
.
.

By processing the 'SETTEXT' message, you can do an `itoa()` (in C) to update a percent complete bar. Of course, you can always just display the message. The HWND you pass SHOULD have a WM_SETTEXT capability. In VB, the EDIT BOX has this capability. An Edit1_change() event will be generated (in VB) each time MF sends you a new number.

The mfReIndex will CLOSE all the indexes and completely recreate them. At the end of indexing, it will reopen your database. The return value is the new DB handle. More than likely, this will be the same DB handle you passed to it. However, it may not be. In either case, you should reassign the return value to the DB handle you passed to it.

e.g. (in C)

```
hMyDB = mfReIndex(NULL, iTask, hMyDB);
if (hMyDB < 0)
      MessageBox(NULL, "Reindexing failed", "WARNING", MB_OK);
```

The return value of mfReIndex WILL be a positive DB handle if it is successful. Otherwise, it will return a negative value indicating the error that occured.

NO other users should access the database while a reindex is being performed.

mfSeek

mfSeek seeks the key specified in the index # specified. All seeks will be 'SOFT' seeks, meaning they will stop at the MATCHING key or the next key HIGHER than the key specified.

e.g.
If you seek for SMITH, seek will return the first record that matches SMITH or the next highest key (like SMYTHE).

```
lDBRecNo = mfSeek( &aKeyToFind, &iCodeReturned, iTASK, iDBHndl, iXHndl)
```

parameters:

- Seek key	Pass the key that you wish to locate in an index
- Code Returned:	MFSEEK_EXACT_MATCH - Found EXACT match, MFSEEK_PARTIAL_MATCH - Found > than
- Your TASK ID	(returned by mfInit)
- The DB Handle for this database	(returned by mfOpen)
- the INDEX # to use	(starts at 0 up to 9 -- Ordinal based on the order in which you created the databases)

returns: (long) Record Number in database that matches or almost matches or EOF (No key was greater than or equal to the key specified...).

Code Fragment (VB):

```
dim lrecordFound as long
dim sName as string * 20
dim icode as integer

sName = "BROWN"
lrecordFound = mfSeek(sName, icode, iTask, iactiveDB, iactiveIDX)

if lrecordFound > 0 then
      if icode =MFSEEK_EXACTMATCH then
            msgbox "Name exists at record " + str$(lrecordFound)
      else
            msgbox "Closest name greater is at record>>
            +str$(lrecordFound)
      endif
else
      msgbox "Database is EMPTY or NO records were greater than key >>
            specified"
endif
```

WARNING:
The key you pass for seeking SHOULD be as large as the index field you are seeking in. Generally, nothing will go wrong if it is too small, but you MAY experience a GPF if you happen to be near a segment boundary...(e.g. if you defined the index to be 20 characters long, make sure you pass a seek string at least 20 characters long).

VB NOTE:
VB has two functions for this function: mfSeekO and mfSeekS.

Since VB doesn't deal with 'pointer' data types too well, we have to trick it into doing what we want. Use mfSeekS when searching on a 'string' index and mfSeekO when searching on ANY other type of index.

mfSkip

Returns next/previous record in index order.

```
lDBRecNo = mfSkip(lFromRecord, iNumberToSkip, iTASK, iDBHndl, iXHndl)
```

parameters:
- From Record Record to skip from
- Number of records to skip +/- # of records to skip (in index order)
- TaskID
- Database to skip in
- Index to base skips on

returns: Next/Previous record # or an error condition (like, EOF/BOF)

code fragment:

```
' This demonstrates processing a file in index order from the TOP -
' to the Bottom of the file...

Dim lCurrentRecord as Long

' Get the TOP (first) record in this index
lCurrentRecord = mfTop(iTask, iActiveDB, iActiveIDX)
do while lCurrentRecord >0    ' Will break on ANY error code (EOF, BOF,
                              ' other-error)

      mRead(lCurrentRecord, aRecordStructure, iTask, iActiveDB, >>
            iActiveIDX, MFRW_ALL)

      ' process record here
      ' .
      ' .
      ' .

      ' Get the next record in the index (1 = next record,
      ' We could go -1 (previous record) or 10 to skip 10 records
      ' in the index...
      lCurrentRecord = mfSkip(lCurrentRecord, 1, iTask, iActiveDB,>>
                              iActiveIDX)
loop
```

mfTop

Returns top (first) record in index order.

```
lTopRecord = mfTop(iTASK, iDBHndl, iXHndl)
```

parameters:
- Task (hMF_TASK) Task from mfInit
- Database (hMF_DB) Database handle from mfOpen
- Index Number (int) Index that you want the 'first' record from

returns: (RPTR) first record in index order

See also: *mfBottom, mfSkip, mfSeek*

If you want to start at the top of an index and read through all the records in the database in 'index' order, you can call this to get the first record in the index, then call mfSkip to move sequentially through the database.

Note that TOP is not the first record of the database, but the first in the index you specify. (i.e. the FIRST record in sorted order). If you want the FIRST record in the database, it is record 1. **Note, however, that Record 1 MAY be deleted! DO NOT try to read a deleted record.** (see mfIsDeleted to check whether a record has been deleted.)

See mfSkip for example

mfUnLock

(release 1.02)

Unlocks a record that has been previously locked.

iStatus = mfUnLock(lRecordNumber, iTask, iDBHndl)

parameters:

```
- Record    Record # to lock
- TASK            Your task number
- DB        The database handle the record is in
```

returns:

```
0  - Unlocked previous lock,
-1 - You either DIDN'T lock the record or something terrible has happened.
```

see also:

mfLock

You must maintain the lock and unlock status of all records.
If you lock a record -- be sure to call UNLOCK. If you do not, that record will remain locked until:
- You close all the databases
- The user disconnects from the network
- The user quits windows

You may have as many records locked as your operating system will allow. Generally speaking, you should only lock one record at a time. The typical time to do so is if a user requests to edit the data in a particular record. You would lock the record until the user finished their changes and then write the record and unlock it for others to use.

Others users may READ a record while it is locked, however, they can not WRITE to it (or lock it themselves...).

mfWrite

Writes/Replaces an existing record.

```
iStatus = mfWrite(lRecord, &sFill, iTASK, iDBHndl, iOption )
```

parameters:

- Record # to write	Record # in the database that you want to write into
- Data buffer	a buffer (YOU created using a structure/'type') to write from
- Task	Task returned from mfInit
- DB	DB returned from mfOpen
- Option	MFRW_ALL, MFRW_DATA, MFRW_KEY

returns: 0 - All OK, -Err occurred.

mfWrite works exactly the same as mfRead, only, in reverse. See mfRead for full parameter explanations.

Note: Before you 'write' to a record, you must create a record. Use mfAppendData to create the initial record.

PERFORMANCE CONSIDERATIONS:

When tuning performance, if you write only the DATA (MFRW_DATA) part of the record, it will be MUCH faster. The slowest part of any database operation is updating the indexes. ANY TIME you can avoid accessing an index, you will get a 90-99% performance increase.
(i.e. you could MFRW_DATA 90 times for every MFRW_ALL or MFRW_KEY).

Severe-Performance API

These functions perform a list of 'transactions' at a very high speed.

mfAppendList

Streams a whole list of items into a database

<Not available in this release>

mfReadList

Reads a list of record pointers from the database index using an index filter or sequentially.
The mfReadList function performs the equivalent of mfSkip, only, it is much faster.
Note: This is NOT similar to most 'filter's you may be accustomed to. It is VERY fast (unlike dbf style filters), but it can only filter 'indexed' keys.

```
lNumberRead = mfReadList(lRecord, &sFilterKey, iFuzzyLength, &lHitListArray,>>
                         lMaxHitsWanted, iTask, iDBHndl, iIDXNum)
```

paramters:

- Record — (Optional) - used for an 'OFFSET' when doing a continuation filter list
  (Required) - used as a starting record when loading in sequential order
- Filterkey — (Optional) - used to load all records (or MaxHitsWanted) records matching filter key. If a record is not specified, this 'seeks' to the first matching record before loading the list.
- Fuzzy — -1 for EXACT key match or length of key to match. REQUIRED for FilterKey
- HitList — - Array of longs that will be filled by the ReadList call. These longs (RPTR)'s are record pointers (i.e. physical record numbers that can be used with mfRead)
- MaxHits — - The maximum # of hits that can be placed in the array. e.g. If you DIM your array to 1000, but there are MORE than 1000 hits, then it will stop filling at 1000.
  - (Optional) MF_SP_COUNT (-1) Just returns a count of records that match (does NOT fill the array)
- Task — - Your task ID
- DB — - The database to use
- Index — - The index to load sequentially from

returns: (long) Number of hits loaded into your array. (or, COUNT of records matching your request)

This function is 1000 times faster than an equivalent:

```
do while curRec > 0
      curRec = mfSkip(...)
loop
```

(A sample is provided in the BCards demo).

mfReadList behaves in five different ways:
1 - Fill a 'filtered' wild-card (fuzzy) list of items in the index that match a filterKey
2 - Fill an EXACT match list
3 - Fill a record-sequential list of records
4 - Fill a continuation list for fuzzy or record sequential lists
5 - Get a count of records that match a criteria

NOTE: These examples use VB syntax. VB auto-converts to the type required for the DLL (specified in the VB DLL Declare statements..). If you are a C person and are wondering WHY we aren't passing by pointer, we actually are. Notice that we use mfReadList0, mfReadListS, and mfReadListNull. These are not functions in the DLL, these are 'DLL redefinitions' in VB. Since C allows you to pass whatever you wish, you can just use mfReadList and pass the correct parms. VB requires us to work-around its problem of not supporting pointers.

1- One of the most interesting features of this is to get a 'filtered' list of items.
e.g. Lets say you have an array of int's as the index item. You want ALL items in the list that have an initial integer value of 2.
VB
```
dim srchArray(0 to 2) as tIntArray
srchArray(0) = 2
lNumberRead = mfReadListO(0, srchArray, 2, &myBigArray, 1000, iTask, >>
                            iDBHndl, iIDXNum)
```

e.g. Lets say you want all the last names that begin with a 'B' in a character index
```
lNumberRead = mfReadListS(0, "B", 1, &myBigArray, 1000, iTask, iDBHndl, iIDXNum)
```

Note that by changing the value of the 'fuzzy length' (parameter 3), we tell it to only compare the FIRST character in the list to our index filter.

C
```
// Loading integers
int   iArray[3];
long  myBigArray[1000];

iArray[0] = 2;
lNumberRead = mfReadList(0, (FPDATA)iArray, 2, (long FAR *)myBigArray, 1000, >>
                        iTask,      iDBHndl, iIDXNum)
```

2- If you wanted all keys in an EXACT match sequence you could say:

```
lNumberRead = mfReadListS(0, "Smith", -1, &myBigArray, 1000,iTask,iDBHndl,iIDXNum)
```

The 3rd parameter (-1) tells the mfReadList function you want EXACT matches ONLY.
NOTE: BE SURE you PAD to the maximum length of the string you are seeking as an exact match. Otherwise, you may get errors or no-hits. If you are using this for 'ints' or 'longs' padding is never required, however, you must still pass all relevant pieces of the 'int/long' index array.

3- Record Sequential
Future releases of MF will include a CC (custom control) that will support a 'browse' table. If you want to be able to display a 'browse' list of all records in the database TODAY, you could use this call to load a group of records in sequential order so that the user can see a nice sorted list. Other than that, it isn't very usefull...
e.g.
VB
```
' This will load the first 1000 records (pointers! to records)
' for the specified DB...
lTopRecord = mfTop(iTask, iDBHndl, iIDXNUM)
lNumberRead = mfReadListNULL(lTopRecord, H0&,0, &myBigArray, 1000, iTask, iDBHndl,>>
                                    iIDXNum)
```

4- Continuation of lists of records
Since a database may have millions of records, it would be silly to think that you can fit all the record pointers into available RAM. Therefore, you can do a 'continuation' list:

With fuzzy/exact lists:
```
lNumberRead = mfReadListO(0, 2, 2, &myBigArray, 1000, iTask, iDBHndl, iIDXNum)
more = True
```

```
do while more
      .
      .
      . process this lNumberRead number of records

      ' ReadList always gives you as many as you ask for (unless there aren't that
      ' many...)
      if lNumberRead < 1000 then
            more = False
      else
      ' Skip 1 record (so we start processing with the NEXT highest record...)
      lNextRec = mfSkip(myBigArray(1000).l,1, iTask, iDHndl, iIDXNum)
      lNumberRead = mfReadListO(lNextRec,2,2,&myBigArray,1000,iTask,iDBHndl,iIDXNum)

      endif

loop
```

Note: In this example, we use parameter 1 with something OTHER THAN 0. This causes mfReadList to start processing with the NEXT record in the list.

With SEQUENTIAL record lists:
substitute:
```
lNumberRead = mfReadListO(lNextRec,2,2,&myBigArray,1000,iTask,iDBHndl,iIDXNum)
```
with:
```
lNumberRead = mfReadListNULL(lNextRec,0,0,&myBigArray,1000,iTask,iDBHndl,iIDXNum)
```

Also, you should pre-calculate the TOP record for the initial read (mfTop(...)).

Creating User Defined Keys

You can only create UDK's (User Defined Keys) in a 'DLL'. If we figure out a way to allow VB to create a UDK (that's fast enough), we'll do it. That said, read on...

A UDK will receive 4 parameters: ptrToMem1, ptrToMem2, the length of the key, and the 'CODE' used when creating the index initially.

Your job is to evaluate the two keys and return either:
-1 Key 1 > Key 2
0 Keys are =
1 Key 1 < Key 2

The name of the **_export**ed function must be '**mfUDK**'. You can place it in a DLL you already have or create a new DLL just for this.

When the you call mfInit, you should pass the filename of the 'DLL' that you placed the function 'mfUDK'. The VB sample bCard*.* uses the reverse-sorting index for the 'Company Name' field. It also shows the proper way to get the UDK DLL loaded.

Code Fragment:
(This code is available as CSAMPLE\UDK\mfUDK.C)

```
/*

  MF.DLL will call this function and pass the following parameters:
  pass:
     val 1   (LPSTR a)
     val 2   (LPSTR b)
     length  (Size of keys to compare)
     type    (This will be a number >= 100)

  the 'type' is the 'type of index' specified when the index's where
  created.  (i.e. When you called mfCreateDB you passed an array of
  index types.  One of those 'types' was >= MFCOMP_UDK.  Whatever that
  # was, it is now passed to you (as the type).  This allows you to
  support multiple UDK's in one function.

  You should return:
  returns:
     0 == equal
     1 == val 1 > val 2
     -1 == val 1 < val 2
Demonstrated in this example is a reverse-sorting for character fields
and a variable length char and INT field.  The 'type' passed for
these fields is:
     1001 = reverse sorting characters
     1002 = variable length char and INT


*/
int FAR PASCAL _export mfUDK(LPSTR a, LPSTR b, int len, int type)
{
  int iReturnValue;
```

```
  switch(type){
     case 1001:
           // _fmemcmp is a MS-C runtime library routine.  (It is
           // actually pretty quick, believe it or not...)
           // Anycase, it returns the EXACT same parameters we need to
           // return from this function...
           // NOTE: To show the reverse-sorting, we just switched the
           // order of the parameters to fmemcmp.  For 'alphabetical'
           // we would have put 'a' and then 'b'.
        return(_fmemcmp((LPSTR)b, (LPSTR)a, len));
        break;

     case 1002:
           // NOTE: By using the value of  'len' we can make this
           // a 'variable' length string routine.  e.g. if the user
           // made the key 80 bytes, this routine would still
           // work (by comparing the first 78 bytes and then the
           // integer tacked onto the end...)
        iReturnValue = _fmemcmp((LPSTR)a, (LPSTR)b, len-2);

           // We can stop checking now because the keys first set of
           // keys (the char[len]) doesn't match, therefore the value
           // of the INT is irrelevant)
        if (iReturnValue != 0)
              return (iReturnValue);

            // NOTE: This may FAIL if the user is using a mfReadList
            // and specifies a shortend key length for the filter.
        if (*(int FAR *)(a+len-2) < *(int FAR *)(b+len-2))
           return (-1);
        if (*(int FAR *)(a+len-2) > *(int FAR *)(b+len-2))
           return (1);

        return(0);  // exact match...
        break;
      case 1003:
            // Add more user defined keys if you wish...
            break;
  }

}
```

NOTES and WARNINGS:
We will attempt to make all severe-performance functions work seemlessly with the UDK's. If we cannot, we will extend the UDK parameter list.

One word of caution: On the mfReadList, a 'fuzzy' search will call your function with a 'shortened' length. (e.g. The key may be 20 bytes, but it will tell you there are only 2 bytes...). In this situation, it is IMPERATIVE that you DO NOT assume the length of the key minus 2 contains an 'integer' value or some other such non-sense...(Our example is an example of what NOT to assume...(case 1002:...))

Therefore, just 'think' about any implications that may arrise from a shortened key and you should not have any problems.

Appendix A
Designing databases for performance

This is a topic that gets dismissed in many manuals. Even the 'books' in a store don't do justice to this topic. (We know, when designing MF we bought dozens of books on these subjects...None of them made much sense. Hopefully, this will.)

READING DATA

Data can be small or big, but the size of it has little to do with the performance of the database. For the most part, it doesn't take any longer to READ 512 bytes than it does 10-bytes. (But, it takes significantly longer to read 50 10-byte records than 1 512 byte record...). There's sort of a reason for this: When the data starts to pump, it can pump 512 just as efficiently as 10. On some machines, data 'packets' can be 4096-bytes. Rarely is a packet any larger than 4096. We have optimized MF for 512-byte packets. This is a 'common' maximum size for a network packet and seems to work well (speed wise). Some networks support 1K, or 4K packets, but don't fret, it's better to be 'under' their maximum packet size than a couple of bytes over (which would generate a second packet of information for a couple of bytes...). Bigger network packets are good if you deal with LARGE data fields or BLOBS, but are wasteful when dealing with index's.

Future releases of MF will contain packet-size optimization and BLOBS.

INDEXES

The SLOWEST part of any database is its' index. Many have forgotten this or don't know about it. Generally, index's are 1/100th as fast as straight data access. AVOID needless indexes. It MAY be faster to sort a 100 record file each time you access it than it is to create an index file for it. However, indexes are convenient and many of us use them just for that reason. If you plan to access a 'small' table very frequently, consider storing it in an array in RAM and writing any data changes to the database without any indexes. (Or, just use the index as the 'sorter' and load the entire table into RAM one time...)

The Severe-Performance functions help resolve some of the problems with indexes. One way is by caching all the index information so no disk read/writes are required. Another way is by loading a bunch of related 'nodes' in the index so that sequential data access is much faster. The problem with caching-indexes in a network environment is that any change to any part of the index can affect the entire index. So, the cached portion of the index is 'dirty' and cannot be used. Additionally, there is no way to know if a particular 'node' in the index has been affected by another user. This problem is what makes 'Client/Server' databases faster than non-C/S databases. A C/S database can cache an index and not worry about someone 'changing' a node in the index. Of course, not everyone needs (or can afford) C/S databases.

In any case, what the Severe-Performance (SP) functions do is: Lock the index momentarily, do their business (in C/S type of style) and then unlock the index for others to access it. Generally, it only takes a 'blip' to do its business. The SP functions stream a whole bunch of records at once into (or out of) the database and free it up again.

TRICKS TO SELLING YOUR PRODUCT and USING (hopefully) MF TO DO IT

BATCHING DATA

You can accomplish some pretty interesting tricks using the severe performance (SP) functions or

just straight accesses. For instance, consider 'BATCHing' in a large quantity of records. Use a regular flat database with no-indexes and then, once per day or as needed, stream all the data in the 'batch' database into the on-line database (the one that has the indexes, etc...).

The performance (to the user) will seem MUCH faster and you will probably be able to handle thousands of more transactions per day than otherwise. This is a trick that most mainframe's use to make them appear faster.

e.g. Say you have 5 mainframe's for data-entry and each mainframe uses it's own 'batch' file. At night, you take all the batches and combine them into the 'on-line' mainframe database. Using this method, you can process many more records at one time and provide faster user response. If you had just one monolithic sized mainframe, you couldn't support as many users (simultaneously) as you could with 5 (or more) cheaper mainframes. Some call this 'distributed processing'. We call it common sense.

If you design your system to utilize this kind of process, you will be able to get it as big as you care to. Think about a system that DOESN'T do this. Lets say, for instance, you work for a mail-order company. As the company grows, you find that the little system that takes orders over the LAN is bogged down. You've hit 100+ order takers and the response time is in the 5 to 10 second range. Now, if the system used a 'batch' type process, you could just add another file server, split the LAN into 2 (or more) separate legs and process the orders at night (or when convenient...). Additionally, you can keep splitting the LAN as the company gets bigger and bigger.

What happens when the biggest, baddest machine can't batch in all the orders during the overnite process? See the next section...

DESIGNING FOR THE FUTURE

Lets say that your biggest machine can't handle all the transactions for the day. Your boss is pissed and about to fire you... What do you do? Split it up. Who says that all the customers have to be in one database? Find the logical separation point and split the data into two (or more) separate systems. Sure doing 'daily totals' will require a little more coding, but don't dismiss this idea yet.

e.g. [true story] A major long-distance company experienced massive growth in the early '80's... (guess who?) In any case, they had the biggest, baddest IBM mainframe you could own. The daily 'call processing and billing' took 23 HOURS to calculate. Not bad. They had one hour to spare... Then, as months went by, the processing time started rising up to 25 HOURS to calculate. They couldn't process all the days calls in one day... The LD company called IBM and had them ship super high speed drives, more RAM, even faster BETA chip-sets... To no avail, the transactions where taking 25-30 hours a day. So, what did they do? They bought 2 more mainframes, put half the customers on one, put half the customers on the other and used one for calculating the daily roll-ups... Not only did this solve thier problem, it gave them a processing time of only 12 hours and room to spare if anything went wrong during the processing...

Transaction Tracking (TTS) and the real problem

This issue has been eroded to the point that most of the articles that cover it don't seem to have any idea about what it really is or haven't ever even tried to implement it. If you're tired of seeing that a database has TTS and all you need to do is 'write to the TTS system and it will automatically rollback any data-loss', etc. and you wonder exactly HOW this works.... Well, anyone can implement TTS by creating a 'batch' area that either ALL process's into the system, or if something crashes, removes the partially added records. That's a nifty feature to have built-in, but what REALLY happens is that you end up with a 'partially' corrupted record or index. What TTS WON'T do,

in every case we've seen, is fix a partially corrupted index or record. Sure, you can re-build the index from scratch. How long do you think that would take on a database, with say, 10 million records? WAY to long. It would be quicker to restore from a back-up.

With that in mind, lets take the issue of 'performance'. If you NEED high-performance, you DON'T want transaction tracking. (OK, yes, in an ideal world, transaction tracking and high-performance go hand-in-hand...but, this isn't an ideal world and we don't all have $20,000 per data server to blow...(on software alone...)). Do mainframes offer TTS and high-speed? Yes, of course they do. But, the controller on a mainframe hard-drive costs $15,000 (as opposed to IDE for $15 bucks and THAT includes a serial/parallel and game port...<g>). Also, storage runs $10,000/gigabyte on a mainframe <plech>.

So, what is TRANSACTION TRACKING and 'ROLLBACK' and HOW do I implement it on my own? Well, you can buy a C/S database that offers it. Or, read on:

Using the BATCH concept, think about how you can implement TTS. The general methodology is:
- Back-up the on-line database
- BATCH in the daily data
- If there is an error (POWER failure, Disk Failure, etc), RESTORE the on-line DB and repeat batch load.

Sound terrible? Yeah, it does. But, think about it. Any data worth protecting is going to be backed up daily (or even HOURLY), right? OK, so A) You made the back-up for the day and B) you gained the performance benefit of the batch process. Not only that, but there wasn't any TTS overhead, so your transactions during the day are faster.

I know SOMEONE has to be thinking, 'What about that BATCH file that was created throughout the day? What if the power fails during a write to it?'. Good question, and the answer is: It's a FLAT file so you can doctor it without worrying about the linkages that will be created later in the day in the on-line database. I realize that you wont always be on-site to correct problems to your application so you should consider a 'doctoring' program that will remove any bad writes (or, at least ignore bad-records when doing the batch process).

Anyway, we hope this information will help you in explaining to your clients WHY your system works better. If you need to discuss implementation issues, please, feel free to contact us.

Additionally, if you don't use MF, all this information still applies to other databases. We just thought it was our job to tell you about it. Heck, why did you buy a database? Because you didn't want to deal with designing the database algorithms. But, you buy it and then no one tells you the tricks to really use it...

DOCTORING

We mentioned doctoring the database in the TTS section. I guess we should tell you HOW you might go about doing this. It's really simple:

MF databases have a 50 byte offset. After that, all the data in the record EXACTLY matches the data structure you have defined it to be (plus 8 'status' bytes appended to the end of each record). Doctoring indexes will have to be left to another discussion...(maybe a future tech note if anyone is truly interested...).

The pertinent header information for an MF DB is:

Number of records in DB: 14th byte (long)

Pointer to 1st deleted record: 18th byte (long)

To convert either of these longs to a logical record position, use:
(lNumberOfRecord * (sizeof(RecordStructure) + 8)) + 50;

WARNING: While you may want to circumvent MF to do read/write, don't do it with any MF database open... The READ's are pretty safe, but if you write to a record, the index tables will NOT be updated and you will end-up with multiple seek pointers to one record. While this could be considered a 'feature', it certainly will confuse you if it is not what you intended to do. A future release will take advantage of this and allow a linked-list to be generated for a single record. (i.e. multiple-keys for one key field, or better known as a 'one to many relationship...')

One nifty feature of this, though, is streaming a list of 'record number sequential' records into RAM. You could open the database and read hundreds of records in one disk I/O. They won't be sorted, but, if you don't care, then feel free.

Appendix B
LIMITS

DATABASE:

Max Record Size:	MAX_INT_VAL (32kish) minus Size of Index
Max DB Size:	2 Gigabytes (unverified...)
Max # Records:	2 Billionish or 2 Gigabytes in total disk space (whichever is first)

INDEX:

Max Key size:	128 Bytes
Max Index Size:	2 Gigabytes
Max # of Records:	(2GB / (KEY_SIZE +9Bytes)) (minus *UP TO 50%* depending on order in which record keys are recieved and indexed) -- This is another common occurence that most manuals FAIL to mention. Anycase, to give you an idea, a 50 BYTE key would top out at 10-20 million records...
Indexing method:	B-Treeish

NOTE: You're index will, inevitably, exceed the size of your database. Unfortunately, there isn't much we can do about this problem. (Actually, your index will only exceed the DB size if you have a rather large index field and a rather small DB field...). In either case, when calculating how many records you can get into a DB, it will usually be MUCH LESS than the maximum values specified. The 'MINIMUM' size of an index key is approximately 9 Bytes + the size of the index key. Don't fret, yet. This is a 'common' occurrence in the DB world. We doubt HIGHLY that other DB's can do it in less space. Future versions may split index files for you. We think all databases should split index files for you, but, that isn't the case, yet. This information is purely for your education.

An interesting note: Is it 'faster' to load data into the index in alphabetical (keyAbetical?) order? No. As a matter of fact, in this DB, it would be SLOWER and the index would be BIGGER. MF indexes don't rely on 'random' data, however, they are more efficient if the data is 'random'. Why is this? To create room in the index, we 'split' nodes to make room for more data. If the nodes fill in sequential order, they will split by 50% each time. To save processing time, we only 'join' nodes if data drops below 50%. If the data is random, it will 'fill' these 'low-value' nodes and no node-splitting will need to occur (unless, of course, the node hits capacity...).

Appendix C
QUICK REFERENCE

I/O
```
int  mfWrite(RPTR Record, FPDATA dFill, hMF_TASK Task, hMF_DB dbHndl, int Option )
int  mfRead (RPTR Record, FPDATA dFill, hMF_TASK Task, hMF_DB dbHndl, int Option )
int  mfDelete(RPTR Record, hMF_TASK Task, hMF_DB dbHndl)
RPTR mfAppendData(FPDATA dFill, hMF_TASK Task, hMF_DB dbHndl)
```

Movement
```
RPTR mfSeek(FPDATA SkStr, int FAR * Code, hMF_TASK Task, hMF_DB dbHndl, int iHndl)
RPTR mfSkip(RPTR record, long NumSkip, hMF_TASK Task, hMF_DB dbHndl, int idxNumber)
RPTR mfTop(hMF_TASK Task, hMF_DB dbHndl, int idxNumber)
RPTR mfBottom(hMF_TASK Task, hMF_DB dbHndl, int idxNumber)
```

Creation
```
int mfCreateIndex(LPSTR FileName, int RecSize, int DataType, int NumOfIndexes)
int mfCreateDB(   LPSTR FileName, int dRecSize,
                  int NumIndex, int FAR * iRecSize, int FAR * iType)
```

General stuff
```
int mfInfoDB(int FAR * RecSize, int FAR * NumIndexs, RPTR FAR * NumRecs,
            RPTR FAR * LiveRecs, hMF_TASK Task,hMF_DB dbHndl)
int mfInfoIndex(hMF_TASK Task, hMF_DB dbHndl, int iHndl)
int mfIsDeleted(RPTR RecordNumber, hMF_TASK TASK, hMF dbHndl)
hMF_DB mfReIndex (HWND HwndStatusDisplay, hMF_TASK iTASK, hMF_DB iDBHndl)
int mfLock(RPTR Record, hMF_TASK Task, hMF_DB iDBHndl)
int mfUnLock(RPTR Record, hMF_TASK Task, hMF_DB iDBHndl)
```

Administrative
```
hMF_DB mfOpen( LPSTR FileName, hMF_TASK Task)
int mfClose(hMF_TASK Task, hMF_DB dbHndl)
hMF_TASK int mfInit( ptmfExtDLL szExtCalls )
int mfDeInit(hMF_TASK Task)
```

Severe-performance functions
```
long mfReadList(RPTR record, FPDATA passStr, int FuzzyMatch, RPTR FAR * hitList,
                  long MaxNumHitsWanted, hMF_TASK Task, hMF_DB dbHndl, int iHndl)
```

Appendix D
ERROR CODES

Error #	Description
Seek/Skip Errors (See also mfSkip for additional return codes)	
-1	mfERR_BOF Technically, this isn't an error. But, if you get this error it means: - You tried to mfTop with NO data in the database - You tried to Skip(-1) and there weren't any more places to skip to -
-2	mfERR_EOF - There was no data in the database on a SEEK - The KEY specified in the SEEK was greater than any other key in the database - You tried to Skip(1) and there weren't anymore places to skip to.
mfOpen	
-10	mfERR_OPEN_UNDEFINED DOS wouldn't open the file. No reason could be determined
-11	mfERR_OPEN_NOHNDLS Out of file handles. Close some other files to open more.
-12	mfERR_OPEN_INVALIDFILE Bad filename passed for database. If the PATH or filename is not valid, this will come back.
-14	mfERR_OPEN_INVALIDFORMAT Attempted to open a file that was not created with this release of MF
-15	mfERR_OPEN_NO_DBINIFILE Undefined in this release
-16	mfERR_OPEN_UDK_NOT_FOUND Couldn't load the UDK dll you specified for the OPEN.
-17	mfERR_OPEN_INDEXMISSING Index file could not be opened. (Not found or otherwise...)
mfClose	
-21	mfERR_CLOSE_BAD More than likely, someone deleted the file while it was open. (Or, maybe the network connection was lost, etc...)
mfWrite	
-31	mfERR_WRITE_BADRECORD Invalid record # passed. Generally, only a negative record # will generate this error. End-of-file is NOT checked for, therefore, you could read millions of records past the end and not generate an error.

-32 mfERR_WRITE_NOLOCK
Couldn't lock the index's for updating. In most cases, you will not experience this error. MF will attempt to lock the index for 20 seconds before it generates this error. If, in that time, it cannot lock the record, it will return this error. If you have a particularly busy system, you may reattempt to lock/write the record.

mfRead

-41 mfERR_READ_BADRECORD
Invalid record # passed. Generally, only a negative record # will generate this error. End of file is NOT checked for, therefore, you could read millions of records past the end and not generate an error.

mfAppendData

-51 mfERR_APPEND_NOADD
Database is full or corrupted.

mfInit

-61 mfERR_REG_NOTASKS
All tasks have been allocated. No more where available for you.

mfCreateDB

-71 mfERR_CREATEDB_BADFILE
Bad filename passed for database. If the PATH or filename is not valid, this will come back. A database name must be a valid DOS filename and must NOT contain an extension.

-72 mfERR_CREATEDB_BADRECSIZE
Attempted to create a database with less than 4 total bytes per record. The database WILL be created, however, if you call mfDelete at a later date, you will GPF. (e.g. You may have a database with less than 4 bytes/record, however, you may not use the mfDelete function on it.)

mfCreateIndex

-81 mfERR_CREATEINDEX_BADFILE
Bad filename passed for database. If the PATH or filename is not valid, this will come back

General Errors (May be caused by any function)

-220 mfERR_BAD_TASKSELECTOR
Bad TASK handle passed

-221 mfERR_BAD_HNDLSELECTOR
Invalid 'database' handle passed

-222 mfERR_BAD_INDEXSELECTOR
Invalid index # (Index # greater than MAX # of index's)

-223 mfERR_BAD_EVERYTHING
One of the 3 above was hosed. Trying to determine WHICH may have caused a GPF, so we didn't test it...(Usually, it means you passed a negative value... The Task/DB/Index #'s should never be negative... If they are, an

error occurred that wasn't caught.)

-300 mfERR_RECORD_DELETED
You tried to perform an operation on a deleted record.
If you have selected the MF Debug mode, this could be returned from any function that accepts a 'RECORD NUMBER' as a parameter.
In non-debug mode, only mfDelete will return this error.

Undocumented API errors (for internal debugging/testing). If we didn't screw up, you shouldn't get one of these... These could also be caused by a faulty drive (unlikely, but possible...).

-91 mfERR_UPDINDEX_BADSEQ
Old data (index) did not exist where it should have.

-92 mfERR_X_NOOLDKEY
No previous index pointer.

-93 mfERR_X_NOADDKEY
No key specified for add

-101 mfERR_SPLIT_NOROOM
Database (index) was completely filled. (You COULD get this if you filled the database to the max...)

Appendix E
Technical Support

Technical support is available to all registered users by the following methods:

For the latest releases and support questions:
The Programmers Corner BBS (Great BBS; 15 or so lines, rarely a busy signal...)
(301) 596-7692
(410) 995-6873
(301) 621-3424
- Send mail to Carl Brown in the General section (#1) or search the files (area: databases) for MF*.ZIP to find the latest release. Note: Due to 'naming' restrictions, make sure you download the file with the 'latest' date. MF is a free download from Programmers Corner.

Internet: CBROWN@access.digex.net
CompuServe: INTERNET:CBROWN@access.digex.net

- All new releases will be placed in the DBAdvisor forum on CompuServe.
NOTE: Shortly, CompuServe support may be dropped. As the 'Internet' provides inexpensive access, we will be moving to an FTP site for updates. All registered users will be notified when this happens. If CompuServe moves to a different method of billing (i.e. gets cheaper) we will keep the CServe access available. However, we have found that most users have Internet access, so we may drop this rather costly CompuServe support.

Those of you that have not experimented with the Internet, please try it. The going rate (in the Washinton D.C. area) is about $25/month for nearly unlimited access. CompuServe 'Forums' are nice -- but expensive. The Internet provides a similar service and it's next to free. (Sorry if this sounds like a cop out for CServe access, but with our costs escalating at a ridiculous rate we just can't afford it without going to a fee-based technical support plan). Pick up a book called 'The Whole Internet'. It'll guide you through what the Internet is and turn you on to some very affordable options that you may not know about.

- Please try to use a method above. If all else fails or in an emergency, we may be reached at (voice):
(301) 340-9134 (may change in January 1994)
(703) 750-1484 (should stay the same...)

When corresponding, -PLEASE- include the version # you are using (found at the top of this document). It is hard to tell what types of errors you have when we don't know your version #. Also, if you are on an older release, we can inform you of where to get the latest release.

Warranty

No warranty is provided at this time. If you really want to see a 'warranty' sheet, read the back of any major software application. I am sure that it will suffice to cover any questions you have.

However, we will do our best to respond to your questions completely and quickly. We will make every effort to fix any reported bug. And, if it is a MAJOR bug (e.g. data loss may occur...), we will make an effort to contact all users with distribution rights.