

ModBaseTM

Version 3.0
by
John McMonagle

ModBaseTM

release 3.0

by

John McMonagle

and

Donald Fletcher

© Copyright 1986-1993 PMI

All Rights Reserved

Repertoire® is a registered trademark of PMI, Box 8402 Green Bay, Wisconsin 54308; United States Patent and Trademark Office Registration Number 1440520. ModBase is a trademark of Donald G. Fletcher, John McMonagle, and PMI.

Table of Contents

Table of Contents2

Terms of Sale3

ModBase4

Compiler Interface.....5

Overview and Examples6

ChkInd9

DBCopier10

DBFields12

DBIndexes15

DBRelate.....24

DBScreen25

Locks28

MemoFunctions.....31

MiscFunctions34

ModBase336

Scrn2DBF.....46

Index48

Terms of Sale

In purchasing ModBase, you are actually purchasing a license to use the information it contains. You agree to assume full responsibility for any damage that your use of that information may do to you or to anyone with whom you deal. You agree to test any programs written with ModBase thoroughly enough to detect any hidden defects it may contain. You agree that the price of ModBase has been discounted to reflect your assumption of that responsibility.

You agree that the United States and the State of Wisconsin shall have exclusive jurisdiction over any lawsuit concerning ModBase; you agree that Green Bay, Wisconsin, is the proper place of venue for any such suit; and you agree to pay PMI's attorney fees if it prevails in any lawsuit against you. You agree to hold PMI and the authors of ModBase harmless and to indemnify them against losses resulting from any lawsuit against you in which ModBase is involved. You agree that all parts of this agreement are severable, and that if any part of the agreement is deemed unenforceable, all other parts remain in full effect. PMI agrees to supply you with any new versions of ModBase that it may produce in the year following your original purchase for approximately the cost of materials, postage, and handling. Documentation for updates may, at PMI's discretion, will be supplied on disk rather than in printed form. PMI also agrees to replace any original diskette that becomes damaged in shipment or that proves defective within a year of your purchase.

Your purchase of ModBase gives you the right to use only one copy of it at a time. You may copy it for backup purposes, and you may link into your own programs' copies of its implementation modules in compiled (object-code) form. In their compiled, linked forms, you may distribute copies of implementation modules and program modules to others, but you may not in any way make available to others the source code or the unlinked object versions of any implementation module or program module. You may also not use ModBase to produce derivative works that compete with it.

PMI reserves the right to pursue any legal remedy for violations of these terms of sale, including the right to revoke your license to use ModBase at any point, even after your application is finished and shipping.

ModBase

ModBase provides programmers with the tools to access data files created by the industry standard database manager, Borland's dBase III. Although the procedures in the ModBase3 modules produce data files compatible with dBase, they can be used independently. DBIndexes creates and maintains dBase3 compatible index files.

While ModBase does provide lowlevel access to database and index files, the design goal is to provide a high level interface where the details of maintaining multiple indexes and file locks are handled internally. For example: If you use DBIndexes.AddToUpdateList all changes to the database file will be reflected in all Indexfiles that are initialized with this procedure. This is done efficiently, as it takes advantage of ModBases internal buffering of records and determines if each key has been changed without actually reading the Index files. This is particularly useful when many index files are used. Some people use more than a dozen indexes, which would be extremely tedious to change manually. Also ModBase uses an advanced record locking system that only locks records for brief periods substantially reducing the likelihood of being locked out of a database.

The pages that follow contain sections for each of ModBase's modules. The sections are arranged in alphabetical order. Each section sets out the module's definition, and then devotes a paragraph or so to each of the procedures and data structures that the module exports.

Compiler Interface.

ModBase is a compiler independent library. Its source code is in Modula-2, and compiles without change under any of the popular Modula-2 compilers for DOS (e.g., Logitech, JPI, Stony Brook, etc.). Its object code has no dependencies on any compiler's runtime support system, and is completely self-contained. The library has no unresolved external references. As a result, you can use ModBase conveniently with any language compiler that follows Microsoft's parameter-passing and object module linkage conventions.

For the convenience of Modula-2 programmers, ModBase's distribution disks provide compiler-specific object modules for the Logitech and JPI TopSpeed Modula-2 compilers. Users of all other compilers, including Stony Brook Modula-2, can use the MB30STN.LIB file provided on one of the distribution disks. It is usable with programs that use any of Microsoft's five standard memory models except small (small code, small data) and compact (small code, large data).

ModBase was written by Donald Fletcher, of New York City, and John McMonagle, of Green Bay, Wisconsin. Mr. Fletcher is a microcomputer consultant with considerable experience in database management, educational software design, and technical writing. He graduated from Princeton University in 1974 with a BA in Anthropology and Linguistics. Mr. McMonagle runs PMI. He has written several programs for the broadcast industry using Modula-2.

ModBase achieves its object-level compiler independence by using procedure variables (pointers to functions) for accessing run-time support services that normal libraries access directly. Initialization of these procedure variables requires that you call a special compiler initialization procedure near the beginning of the main body of your program. These compiler dependent parts are set in a module called InitCompilerMods that is different for each compiler. Also there is a separate version of NumTypes.def and NumTypes.mod to handle the differences in number representation between the different compilers. It is actually sufficient simply to IMPORT the appropriate Init module into your main module, since the Init module will automatically execute the AssignProcs procedure during the module initialization sequence.

Overview and Examples

Following is AutoMix.mod a simple diagnostic program that is used to test ModBase's Indexes for random changes. In actual use about 4 instances of it were started.

```

MODULE AutoMix;      (* Random changes *)

(*  IMPORT PMD;*)
FROM SYSTEM IMPORT BYTE;
IMPORT InitCompilerMods;
IMPORT Locks;
FROM DBIndexes IMPORT DBIndex, InitIndex, CloseIndex,
    FindPositionCh,
    AddToUpdateList;
FROM DBFields IMPORT Replace;
FROM ModBase3 IMPORT
    DBFile, InitDBF, OpenDBF, ReadDBRec, WriteDB
    Rec,

    NumberRecords, LockRec, UnLockRec, DefaultF
    ixUp, CloseDBF;
FROM ErrorManager IMPORT WARN;
FROM Timing IMPORT BeginTimer, ShowUsage;
FROM Rand IMPORT RandomInit, RandomCard, Randomize;
    (* a variation of logitechs
        random *)
(* FROM Lib IMPORT RANDOM;*)
IMPORT StringIO;
IMPORT StrConv, OS2KBD(*tests were done under os2*);
VAR
    counter:CARDINAL;
    string:ARRAY[0..79] OF CHAR;

PROCEDURE KeyPressed():BOOLEAN;
VAR r:CARDINAL;
k:OS2KBD.CharDataType;
BEGIN
    k.asciiCharCode:=0C;
    k.scanCode:=0;
    k.NLSShiftStatus:=0;
    r :=OS2KBD.KBDPEEK(k,0);
    RETURN (k.scanCode#BYTE(0)) OR (k.asciiCharCode#0C);
END KeyPressed;

PROCEDURE RandStr(VAR S : ARRAY OF CHAR);
(* Generate a string of random letters with a random
    length *)
VAR
    I : CARDINAL;
    Len : CARDINAL;
BEGIN

```

Overview and Examples

```
Len := 1 + (*RandomCard*)RandomCard(HIGH(S)+1) ; (*
    random 1 to 30 Len *)
FOR I := 0 TO Len - 1 DO
    S[I] := CHR((*RandomCard*)RandomCard(26)+65); (*
        random letters *)
END (* for *);
FOR I := Len TO HIGH(S) DO
    S[I] := ' ';
END (* if *);
END RandStr;

CONST
    KeyLen = 30;
    Buffers = 400;

TYPE
    KeyStr = ARRAY [0..KeyLen-1] OF CHAR;

VAR
    DBF : DBFile;
    NDX : DBIndex;
    Name : KeyStr;
    Found : BOOLEAN;
    rec:LONGINT;
    res:CARDINAL;

BEGIN
    Randomize;
    res:=OS2KBD.KBDFLUSHBUFFER(0);
    counter:=0;

    InitDBF('Modbase.DBF',DBF,0,FALSE,FALSE,
        TRUE,DefaultFixUp);
    IF OpenDBF(DBF) THEN END;(* in a real program you
        should make sure the file is
        opened *)
    InitIndex("ModBase.NDX", NDX,DBF,
        Buffers,FALSE,TRUE,FALSE);
    (* as you can see the index is not opened. It will be
        opened automatically
        it is a good idea open it to check for it's
        existence *)
    AddToUpdateList(DBF,NDX);
    REPEAT
        (*
            rec:=VAL (LONGINT,RandomCard(VAL (CARDINAL
                ,NumberRecords(DBF))))+1);*)
        rec:=VAL (LONGINT,RandomCard(10)+1);(*increace odds
            of contention*)
        RandStr(Name);
        ReadDBRec (DBF,rec);
        Locks.Pause(200);
        (* one Pauses to give other instances a chance at
            the file this
            would not normally be nesssicary, but in this case
            access is almost
```



```

        continious *)
Replace(DBF,1,Name);
WriteDBRec(DBF); (* the index will be automatically
                updated *)
StrConv.CardinalToStr(counter,8,string);
StringIO.WriteStr(StringIO.outp,string);
INC(counter);
UNTIL (counter=65000) OR KeyPressed();
CloseIndex(NDX); (* make sure you close files *)
CloseDBF(DBF);
DEC(counter);
res:=OS2KBD.KBDFLUSHBUFFER(0);
END AutoMix.

```

To show how ModBase is used with Repertoire's screen design/display system, your distribution disks also provide an example named ADDRBOOK. It consists of ADDRBOOK.MOD, which contains the Modula-2 source code for the program; ADDRBOOK.EDT, which contains editable screen definitions for use with Repertoire's screen compiler; ADDRBOOK.DSP, which contains compiled screens produced by running ADDRBOOK.EDT through the Repertoire screen compiler; ADDRBOOK.DBF, a data file containing several names and addresses; and ADDRBOOK.DBX, an index for the data file. ADDRBOOK's source code is a bit more sophisticated than CHECKREG.MOD, and it contains several useful procedures, such as ImportFromAscii, ExportToAscii, but the real point of ADDRBOOK is to illustrate the use of ModBase's Scrn2Dbf module, which transfers data automatically between Repertoire's screen system and a ModBase data file. Scrn2Dbf is explained below. Please see Chapter 3 of the Repertoire manual for a complete explanation of Repertoire's screen system. ADDRBOOK also is simple example of how to use initialize create and maintain database and index files.

ChkInd

DEFINITION MODULE ChkInd;

```
( *
 *   ModBase
 *   Release 3.0
 *   By Don Fletcher & John McMonagle
 *   (c) Copyright 1986 - 1991 PMI
 *   P.O. Box 8402
 *   Green Bay WI 54308
 *   All Rights Reserved
 *
 *)
```

FROM DBIndexes IMPORT

FindPositionCh, OpenIndex, BuildIndex, DBIndex,
NextRecord, PrevRecord;

PROCEDURE NDXChk(VAR ndx:DBIndex);

END ChkInd.

ChkInd is a diagnostic module that will test to see if an index is in order and is not missing any keys. It contains only one procedure and is very simple to use; just open an index file using the OpenIndex procedure in the DBIndexes module, and pass the DBIndex variable to NDXChk procedure. It will report any errors that it finds. If it finds no errors at all, it exits without comment.

DBCopier

```
DEFINITION MODULE DBCopier;
```

```
(*
 *   ModBase
 *   Release 3.0
 *   By Don Fletcher & John McMonagle
 *   (c) Copyright 1986 - 1991 PMI
 *   P.O. Box 8402
 *   Green Bay WI 54308
 *   All Rights Reserved
 *)
```

```
FROM ModBase3 IMPORT
    DBFile;
```

```
PROCEDURE DBCopy( VAR Source: DBFile; VAR Dest:
    DBFile; CopyDeleted : BOOLEAN );
```

```
PROCEDURE DBPack( VAR dbf: DBFile );
```

```
PROCEDURE DBFieldCopy( VAR Source: DBFile; VAR Dest:
    DBFile);
```

```
END DBCopier.
```

```
PROCEDURE DBCopy ( VAR Source : DBFile; VAR Dest :
    DBFile; CopyDeleted : BOOLEAN );
```

DBCopier provides an efficient, convenient way to copy the contents of one data file into another. You do not have to worry about copying memo files and fields separately; DBCopier takes care of all that. And it is smart enough to know that when two data files have the same field structure, it does not have to copy records field by field. It copies records as large blocks of memory in that case, which speeds up the operation considerably. DBCopier's last parameter, CopyDeleted, lets you control whether it copies records that have ostensibly been deleted from the source file. Since records are not physically deleted from a dBase file until the file is packed, DBCopier can ignore the fact that a record has been marked deleted, and will do so if you pass TRUE to its CopyDeleted parameter.

```
PROCEDURE DBPack ( VAR dbf : DBFile );
```

DBPack crunches a data file and any associated memo file by copying each of its records into a temporary data file, skipping over records that have been marked for deletion. It then deletes the source file and renames the temporary file to the name originally used by the source file.

```
PROCEDURE DBFieldCopy ( VAR Source : DBFile; VAR Dest
                      : DBFile);
```

DBFieldCopy always works on the current record of both the source DBFile and the destination DBFile. It examines every field in the source file's current record, and if its name matches that of the field at the corresponding position in the destination file's current record, it copies the contents of that field into the destination file's record. It checks for field type mismatches, and it copies memo fields as well as any other kind of field.

DBFields

```
DEFINITION MODULE DBFields;
```

```
(*
*   ModBase
*   Release 3.0
*   By Don Fletcher & John McMonagle
*   (c) Copyright 1986 - 1991 PMI
*   P.O. Box 8402
*   Green Bay WI 54308
*   All Rights Reserved
*
*)
```

```
FROM ModBase3 IMPORT DBFile;
FROM DateFunctions IMPORT Date;
FROM NumTypes IMPORT Real8;
```

```
PROCEDURE DateToDataFormat(d: Date; VAR datestring:
    ARRAY OF CHAR);
```

```
PROCEDURE DataFormatToDate(dbstring: ARRAY OF CHAR;
    VAR d: Date);
```

```
PROCEDURE GetDateField(VAR alias: DBFile; fieldnum:
    CARDINAL; VAR d: Date);
```

```
PROCEDURE GetLogicalField(VAR alias: DBFile;
    fieldnum: CARDINAL; VAR l: BOOLEAN);
```

```
PROCEDURE GetNumField(VAR alias: DBFile; fieldnum:
    CARDINAL; VAR n: Real8);
```

```
PROCEDURE Replace(VAR alias: DBFile; fieldnum:
    CARDINAL; s: ARRAY OF CHAR);
```

```
PROCEDURE Replaced(VAR alias: DBFile; fieldnum:
    CARDINAL; d: Date);
```

```
PROCEDURE ReplaceL(VAR alias: DBFile; fieldnum:
    CARDINAL; l: BOOLEAN);
```

```
PROCEDURE Replacen(VAR alias: DBFile; fieldnum:
    CARDINAL; n: Real8);
```

```
END DBFields.
```

The purpose of DBFields is to provide access and interpretation of fields in dBase3 type files. dBase keeps its data as text strings, and the ModBase3 procedure GetField returns the value of the field in a string. Since the value is often actually a number, or a BOOLEAN value, or a date, it is convenient to be able to interpret the strings in the data file appropriately. DBFields provides a

set of procedures to handle this sort of interpretation. The procedures `GetNumField` and `ReplaceN` interpret numeric values as REALs, but there is no reason why the numeric values could not be interpreted as INTEGER, CARDINAL, or DECIMAL values, if necessary.

These procedures do not validate the values as correct for their types. It is the responsibility of the programmer to ensure, for example, that if a numeric field is specified with a width of 12 and 2 decimal places, the value passed to the `ReplaceN` function does not exceed these limits.

Note that the `Replace` procedures in `DBFields` all act on the current record of the `DBFile` passed to the procedure. They do not write the change to the disk, which must be done by the procedure `WriteDBRec` imported from `ModBase3`.

```
PROCEDURE DateToDataFormat( d: Date; VAR datestring:
                           ARRAY OF CHAR );
PROCEDURE DataFormatToDate( dbstring: ARRAY OF CHAR;
                           VAR d: Date );
```

`DateToDataFormat` converts a date from TYPE `Date` (see `Repertoire's DateFunction's` module) to a string in the form `YYYYMMDD`, which is how `dBase` stores dates. `DataFormatToDate` converts a string in the format `YYYYMMDD` into the `Repertoire` type `Date`.

```
PROCEDURE GetDateField( VAR alias: DBFile; fieldnum:
                       CARDINAL; VAR d: Date );
PROCEDURE GetLogicalField( VAR alias: DBFile;
                          fieldnum: CARDINAL; VAR l: BOOLEAN );
PROCEDURE GetNumField( VAR alias: DBFile; fieldnum:
                      CARDINAL; VAR n: Real8 );
```

`GetDateField` gets a field as a string and interprets it as a date. `GetLogicalField` gets a logical field as a character and interprets it as a `BOOLEAN` variable. `GetNumField` gets a numeric field as a string and interprets it as a real number. `Replace` replaces, in the current record of `alias`, the field specified by `fieldnum` with `s`.

```
PROCEDURE Replace(VAR alias: DBFile; fieldnum:
                 CARDINAL; s: ARRAY OF CHAR);
PROCEDURE Replaced(VAR alias: DBFile; fieldnum:
                  CARDINAL; d: Date);
PROCEDURE ReplaceL(VAR alias: DBFile; fieldnum:
                  CARDINAL; l: BOOLEAN);
PROCEDURE ReplaceN(VAR alias: DBFile; fieldnum:
                  CARDINAL; n: Real8);
```


The procedure `Replace` changes the value of the field identified by `fieldnum` in the DBFile's current record to the value in `s`. Be careful. `Replace` is a generic function; it does not do type checking. ModBase provides three other procedures -`ReplaceD`, `ReplaceL`, and `ReplaceN` -- which provide some type checking. `ReplaceD` changes the value of a date field. `ReplaceL` changes the value of a logical field. `ReplaceN` changes the value of a numeric field. If you have a Fixed point string you can use the untyped `Replace` procedure for speed purposes, but make sure the decimal point is aligned as in the field specification (see `FrameToDBF`).

`Replace` does not write the altered record to the file. The procedure `WriteDBRec` in `ModBase3` writes the current record to the file. `ReplaceD` accepts a variable of Date TYPE and replaces the field specified by 'fieldnum' in current record with a string representing that date. `ReplaceL` accepts a BOOLEAN variable and replaces the field specified by `fieldnum` in the current record with a string representing that value. `ReplaceN` accepts a REAL variable and replaces the field specified by `fieldnum` in current record with a string representing that value.

DBIndexes

DEFINITION MODULE DBIndexes;

```
(*
* ModBase
* Release 3.0
* By Don Fletcher & John McMonagle
* (c) Copyright 1986 - 1991 PMI
* P.O. Box 8402
* Green Bay WI 54308
* All Rights Reserved
*)
```

```
FROM ModBase3 IMPORT DBFile;
FROM NumTypes IMPORT Real8;
```

TYPE

DBIndex;

```
KeyProcedure = PROCEDURE( DBFile, DBIndex ,VAR ARRAY
                        OF
                        CHAR );
```

```
PROCEDURE DefaultKeyProcedure(alias: DBFile; ndx:
                        DBIndex;
                        VAR str:ARRAY OF CHAR );
```

```
PROCEDURE FindPositionCh(ndx: DBIndex; keystr:
                        ARRAY OF CHAR; VAR found: BOOLEAN);
```

```
PROCEDURE FindPositionN( ndx: DBIndex; keystr:
                        ARRAY OF CHAR; VAR found: BOOLEAN);
```

```
PROCEDURE InitIndex(indexname: ARRAY OF CHAR;VAR
                        ndx: DBIndex;alias:DBFile ;bufferize: CARDINAL;
                        safety,IncludeDeleted,exclusive:BOOLEAN);
```

```
PROCEDURE InitCompIndex(indexname: ARRAY OF CHAR; VAR
                        ndx: DBIndex;alias:DBFile;Key :KeyProcedure;
                        bufferize: CARDINAL
                        ; safety,IncludeDeleted,exclusive:BOOLEAN);
```

```
PROCEDURE OpenIndex( ndx: DBIndex):BOOLEAN;
```

```
PROCEDURE DisposeIndex(VAR ndx: DBIndex);
```

```
PROCEDURE InsertEntry( ndx: DBIndex; kstr: ARRAY
                        OF CHAR; recno: LONGINT );
```


ModBase™ 3.0

```
PROCEDURE CloseIndex( ndx: DBIndex);

PROCEDURE UpdateIndex( ndx: DBIndex);

PROCEDURE SetSafetyOn( ndx: DBIndex);

PROCEDURE SetSafetyOff( ndx: DBIndex);

PROCEDURE SetIndexBuffers( ndx:
                          DBIndex;Buffers:CARDINAL);

PROCEDURE NextRecord( ndx: DBIndex; VAR recno:
                      LONGINT): BOOLEAN;

PROCEDURE PrevRecord( ndx: DBIndex; VAR recno:
                      LONGINT): BOOLEAN;

(* Builds may fail because of failure getting
   exclusive access *)
PROCEDURE BuildIndex( ndx: DBIndex; keyexp: ARRAY OF
                      CHAR):CARDINAL;

PROCEDURE BuildCompIndex( ndx: DBIndex; type: CHAR;
                          keyexp:ARRAY OF CHAR;size : CARDINAL):CARDINAL;

PROCEDURE AddRecord( alias: DBFile; ndx:
                      DBIndex);

PROCEDURE CurrentKeyCh( ndx: DBIndex;VAR val:ARRAY OF
                        CHAR);

PROCEDURE CurrentKeyN(ndx: DBIndex):Real8;

PROCEDURE CurrentRec(ndx: DBIndex):LONGINT;

PROCEDURE NumKeyType( ndx: DBIndex):BOOLEAN;

PROCEDURE KeyLength( ndx: DBIndex):CARDINAL;

PROCEDURE DeleteCurrentEntry( ndx: DBIndex);

PROCEDURE GoTop( ndx: DBIndex);

PROCEDURE GoBottom( ndx: DBIndex);

PROCEDURE AddToUpdateList( alias: DBFile; ndx:
                           DBIndex);
(* add ndx to alias's list of indexes to update *)

(* PROCEDURE UpdatedBIndexes( alias:DBFile);
   assigned to Index check procedure in Modbase3 *)

PROCEDURE UpdateUnique( alias: DBFile; ndx:
                        DBIndex):BOOLEAN;
```

```
(* must be called after changes are made and before
WriteDBFile
Will indicate TRUE if key is not changed or the new
value of the key is
not in the index. It will only be false if the Update
will produce
a duplicate.*)
(* you normally never need to call the EnterLock or
ExitLock
even on a network only need if you use InsertEntry which
you should
never need to use *)
PROCEDURE EnterLock( ndx:DBIndex);
PROCEDURE ExitLock( ndx:DBIndex);

END DBIndexes.
```

The purpose of the DBIndexes module is to provide indexed sequential access to dBase3 type files. It is a simple B-Tree type indexing module.

The indexing routines in this module depend on an opaque type called DBIndex. DBIndex is actually a pointer to an internal record which contains information used for housekeeping purposes. One of the things DBIndexes manages in this way is a read-write buffer, which improves performance. The size of the buffer is set when you call the OpenIndex procedure. The buffer is written to disk by CloseIndex.

In releases of ModBase preceding 2.1, the DBIndex type was not opaque. To upgrade from earlier releases to 2.1, therefore, you will have to replace direct references to the contents of DBIndex with calls to equivalent procedures that DBIndexes exports. Consider DBIndexes as unit or object that you just initialize and it takes care of it's self .

Another is an internal record associated with each index that we call the "safety field." The Safety field controls whether ModBase keeps the disk physically up to date at all times. You can control the value of the Safety field with the procedures SetSafetyOn and SetSafetyOff. It is set to TRUE by default. Setting it to FALSE speeds up indexing operations, but if you set it back to TRUE again, be sure to call the UpdateIndex routine first. Otherwise you will lose data. With Safety off, the danger is minimal, but one could lose data if the power failed or the system crashed for some reason. Safety on is considerably slower when adding or deleting keys, but probably worth it in data entry routines. One should always have some provision to rebuild indexes, as one missing entry can effectively cause the loss of a lot of data.

ModBase™ 3.0

```
KeyProcedure = PROCEDURE( DBFile, DBIndex ,VAR ARRAY  
OF CHAR );
```

KeyProcedure is used in **InitCompIndex** to specify how keys are to be built in an index file. You may write your own to define any type of ordering you desire.

```
PROCEDURE DefaultKeyProcedure(alias: DBFile; ndx:  
DBIndex; VAR str:ARRAY OF CHAR );
```

DefaultKeyProcedure is used as the default **KeyProcedure**. It will handle simple indexes based on a single **DBase3** field name.

```
PROCEDURE FindPositionCh(VAR ndx: DBIndex; keystr:  
ARRAY OF CHAR; VAR found: BOOLEAN);  
PROCEDURE FindPositionN(VAR ndx: DBIndex; keystr:  
ARRAY OF CHAR; VAR found: BOOLEAN);
```

Numeric keys are indexed using the procedure **FindPositionN** and all other types are indexed with **FindPositionCh**. All keys other than numeric keys are represented in the index as strings of characters. Numeric keys are represented as 8-byte REALs. **FindPositionN** works just like **FindPositionCh**, but indexes numeric values rather than string values.

FindPositionCh indexes all fields except numeric and date fields. It sets the indexes current key value to the appropriate record number in the data file. If there is no such value, it sets its found parameter to FALSE.

After a call to one of the **FindPosition** routines, you can use the **CurrentRec** procedure to retrieve the record from the data file. For example:

```
ModBase3.ReadDBRec( CustomerData,  
CurrentRec(IndexByZipCode ) );
```

```
PROCEDURE InitIndex(indexname: ARRAY OF CHAR;VAR ndx:  
DBIndex;alias:DBFile ;bufferize:  
CARDINAL; safety, IncludeDeleted,  
exclusive:BOOLEAN);
```

```
PROCEDURE InitCompIndex(indexname: ARRAY OF CHAR; VAR  
ndx:DBIndex; alias:DBFile;  
Key:KeyProcedure; bufferize: CARDINAL;  
safety, IncludeDeleted,  
exclusive:BOOLEAN);
```

```
PROCEDURE OpenIndex( ndx: DBIndex):BOOLEAN;
```

```
PROCEDURE CloseIndex( ndx: DBIndex);
```

In ModBase's 2.1 release, the process of opening an index has been split into two steps. You must first initialize a DBIndex variable with either the InitIndex or the InitCompIndex (for Initialize "Complex" Index) procedures. You then open the index by calling a separate OpenIndex procedure.

OpenIndex opens the index file identified by the name you passed as the first parameter of InitIndex when you initialized the index variable. It reads part or all of the file into a buffer, the size of which was specified by the buffersize parameter when you called InitIndex. Buffersize gives the number of nodes (each of which occupies about 525 bytes in this implementation) which are maintained as a read/write buffer in memory. If you allocate too few buffers, OpenNdx will nevertheless use some minimum number usually about 3. Buffers are only allocated as needed. safety if TRUE will cause it to flush all buffers to the disk after each operation. This is usually not needed and it slows operation considerably. safety is useful if power failures are likely. IncludeDeleted specified if deleted records will be included in the database. exclusive if TRUE will give exclusive access to the index file. If the index is already in use your open will fail. ModBase will automatically open indexes as need. You do not have to call OpenIndex but it is a good idea to call OpenIndex to make sure the index exists. If you are short on handles or memory you can close index files that are not actively being used such as those were the indexes current position is not relevant.

CloseIndex closes the index and writes all buffers to the disk. It is a good idea to use a terminate procedure (see the ErrorManager section of the Repertoire manual) to close your index files if you keep safety off. Otherwise, added keys may be lost if you terminate unexpectedly.

```
PROCEDURE DisposeIndex(VAR ndx: DBIndex);
```

DisposeIndex first calls CloseIndex to write all of ndx's buffers to disk, and then deallocates all of the memory associated with ndx and sets it to NIL.

```
PROCEDURE NextRecord(VAR ndx: DBIndex; VAR recno:
                    LONGINT): BOOLEAN;
PROCEDURE PrevRecord(VAR ndx: DBIndex; VAR recno:
                    LONGINT): BOOLEAN;
```

NextRecord sets its 'recno' parameter to point to the next record in the index. This may or may not be the "next" record in the data file, of course, because the ordering of keys in the index depends on how the index was

built. In an index based on last names of customers, for example, NextRecord might set its recno parameter to 34233 and then to 897 if those records contained data for, say, Jim Smyth and Albert Smythe, respectively. PrevRecord sets recno to point to the previous record in the index. Both procedures return FALSE if there are no more records in the specified direction. This a very fast operation, and usually involves no disk access.

```
PROCEDURE GoTop(VAR ndx: DBIndex);  
PROCEDURE GoBottom(VAR ndx: DBIndex);
```

GoTop and GoBottom are meant to be used in conjunction with NextRecord and PrevRecord. After you call GoTop, NextRecord gives you the number of the data file record associated with the first entry in the index. After you call GoBottom, PrevRecord gives you the number of the data file record associated with the last entry in the index. These are fast and are based on the Btree structure and involve no comparisons and most of the time will not involve any disk activity.

```
PROCEDURE CurrentKeyCh( ndx: DBIndex;VAR val:ARRAY OF  
                        CHAR);
```

```
PROCEDURE CurrentKeyN(ndx: DBIndex):Real8;
```

```
PROCEDURE CurrentRec(ndx: DBIndex):LONGINT;
```

```
PROCEDURE NumKeyType( ndx: DBIndex):BOOLEAN;
```

```
PROCEDURE KeyLength( ndx: DBIndex):CARDINAL;
```

These procedures are meant to provide information about index files that, in earlier versions of ModBase, you had to get by direct access to the DBIndex record, which is now hidden. You should seldom need to use them, because in most programs the information they provide will be obvious from the context. DBIndexes exports them primarily to guard against the possibility that hiding the DBIndex record might reduce the functionality of unusual programs. The names of the procedures make clear what they do.

```
PROCEDURE AddRecord( alias: DBFile; ndx:  
                    DBIndex);
```

AddRecord is the normal way to manually add an index entry. It adds the current record of DBFile alias to Index ndx. If you use AddToUpdateList all changes to the database file will automatically be made to the index file.

```
PROCEDURE InsertEntry(VAR ndx: DBIndex; kstr: ARRAY OF
                      CHAR; recno: LONGINT);
```

InsertEntry is a very low level routine. You should never have to use it. It would allow you to build an Index manually that had no particular data file. The following is a code segment using InsertEntry with a Complex index key. AddToUpdateList would be a much easier solution. Providing automatic updates of the Index file.

```
PROCEDURE KeyByDateAProcedure( VAR dbf :
                               ModBase3.DBFile; VAR str : ARRAY OF CHAR
                               );
```

```
VAR
```

```
    ok      : BOOLEAN;
```

```
    string  : ARRAY[ 0 .. 29 ] OF CHAR;
```

```
BEGIN
```

```
    ModBase3.GetField( dbf, 2, str, ok );
```

```
    ModBase3.GetField( dbf, 3, string, ok );
```

```
    Strings.Concat( str, string, str );
```

```
END KeyByDateAProcedure;
```

```
IF Scrn2DBF.FrameToDBF( Apointment, DspFiles.MainFrame,
                        dumcard) THEN
```

```
END (* if FrameToDBF *);
```

```
ModBase3.WriteDBRec( Apointment );
```

```
KeyByDateAProcedure(Apointment, str );
```

```
DBIndexes.FindPositionCh( ApointmentByDateAE, str,
                          dumbbool );
```

```
DBIndexes.InsertEntry( ApointmentByDateAE, str,
                       Apointment.currentrecnum );
```

```
PROCEDURE BuildIndex( ndx: DBIndex; keyexp: ARRAY OF
                      CHAR):CARDINAL;
```

BuildIndex indexes all the records in alias, overwriting any existing index (or other file) with the same name. BuildIndex closes the index after building it, so the index must be opened again with OpenIndex before use. Indexes are built with exclusive access Build Index will return a non zero value if it fails. The return code is the standard DOS error code.

```
PROCEDURE InitCompIndex(indexname: ARRAY OF CHAR; VAR
                        ndx: DBIndex; alias:DBFile;Key
                        :KeyProcedure; buffersize: CARDINAL
                        ;Safety,IncludeDeleted,exclusive:BOOLEAN
                        );
```



```
PROCEDURE BuildCompIndex( ndx: DBIndex; type: CHAR;
    keyexp:ARRAY OF CHAR;size : CARDINAL):CARDINAL;
```

InitCompIndex and **BuildCompIndex** are for creating complex indexes. To use them, write a procedure that is assignment compatible with **KeyProcedure** and pass it to the key parameter. The type 'C' or 'N' for numeric and date fields. The keyexp parameter is the Dbase expression for the key it allows Dbase3 to use the index if supplied, ModBase does not use it.. The size parameter must designate the length of your key. The purpose of **BuildCompIndex** is to get around **BuildIndex**'s inability to create indexes on other than a simple field. Just write a **KeyProcedure** to create the key you want and pass it as **Key**. The following procedure is an example of a **KeyProcedure**:

```
PROCEDURE DefaultKeyProcedure(VAR alias: DBFile; VAR
    ndx: DBIndex; VAR str:ARRAY OF CHAR );
VAR
    OK:BOOLEAN;
BEGIN
    GetField(alias, ndx^.keyexpression, str);
    CAPstr(str);
END DefaultKeyProcedure;

PROCEDURE AddToUpdateList( VAR alias: DBFile; VAR
    ndx: DBIndex);
```

AddToUpdateList is what allows ModBase to maintain indexes automatically. By calling **AddToUpdateList**, you associate an index file with a data file, so that operations on the data file can be reflected in automatic operations on all index files associated with the data file. Any number of index files can be associated with a single data file.

```
PROCEDURE DeleteCurrentEntry(VAR ndx: DBIndex);
```

DeleteCurrentEntry would normally be used only if the record in question changed the value of its indexed field. **DeleteCurrentEntry** makes no attempt to re balance the tree, so indexes for files whose keys are edited heavily should be rebuilt periodically with **BuildIndex** to ensure maximum efficiency.

```
PROCEDURE UpdateIndex(VAR ndx: DBIndex);
```

UpdateIndex is used when safety is off to make sure an index has been written to disk. If safety is off, **UpdateIndex** will also write all buffers to disk. If you are switching from safety off to safety on, first call this to be sure that everything is up to date.

```
PROCEDURE UpdateUnique( VAR alias: DBFile; VAR ndx:
                        DBIndex):BOOLEAN;
```

UpdateUnique is used to maintain an index in which all values must be unique. To use it, you must call it between and replace operations in the DBF and any updates to the Index. At that point a TRUE means that the update will be unique and FALSE means it will be a duplicate. At that point you can prompt for new input, or take some other action to obtain a unique key.

```
PROCEDURE EnterLock( ndx:DBIndex);
PROCEDURE ExitLock( ndx:DBIndex);
```

EnterLock Locks the index and ExitLock unlocks the index. You should never need to use these procedures. By locking an index you can speed a series of access quite a bit, but this will lock out all other processes.

DBRelate

```
DEFINITION MODULE DBRelate;
```

```
( *
 *   ModBase
 *   Release 3.0
 *   By Don Fletcher & John McMonagle
 *   (c) Copyright 1986 - 1991 PMI
 *   P.O. Box 8402
 *   Green Bay WI 54308
 *   All Rights Reserved
 *
 *)
```

```
FROM ModBase3 IMPORT DBFile;
FROM DBIndexes IMPORT DBIndex;
```

```
PROCEDURE ReadRelate(key : ARRAY OF CHAR; ndx:
    DBIndex; dbf: DBFile): BOOLEAN;
```

```
END DBRelate.
```

```
PROCEDURE ReadRelate(key : ARRAY OF CHAR; ndx:
    DBIndex; dbf: DBFile): BOOLEAN;
```

ReadRelate helps save a few lines if you have relational files in ModBase. For example, you may have a very large database of customers; to save space and data entry time you could use two DBFiles instead of one:

Customer file fields	ZipFile Fields
-----	-----
Name	ZipCode
StreetAddress	City
ZipCode	State

The ZipFile is indexed by ZipCode in file ZipIndex. If you have just read a record from the customer file, you would get the ZipCode, then call **ReadRelate(ZipCode,ZipIndex,ZipFile)**. Then you will have all the information available. **ReadRelate** saves a few lines and makes the operation a bit more readable.

DBScreen

```
DEFINITION MODULE DBScreen;
```

```
( *
 *   ModBase
 *   Release 3.0
 *   By Don Fletcher & John McMonagle
 *   (c) Copyright 1986 - 1991 PMI
 *   P.O. Box 8402
 *   Green Bay WI 54308
 *   All Rights Reserved
 *
 *)
```

```
IMPORT SmartScreen; (* from Repertoire *)
FROM ModBase3 IMPORT DBFile, DBFieldDescriptor;
```

```
TYPE
```

```
    Direction = (Ahead, Nowhere, Back, Escape);
```

```
VAR
```

```
    readattr, writeattr:
```

```
    RECORD
```

```
        fore, back: SmartScreen.Colors;
```

```
        MonoAttr: SmartScreen.TextAttribute;
```

```
    END;
```

```
    lastdirection: Direction;
```

```
PROCEDURE CreatedBF(dbfilename: ARRAY OF CHAR; VAR
    alias: DBFile);
```

```
PROCEDURE GetDescriptors(VAR fields: ARRAY OF
    DBFieldDescriptor);
```

```
PROCEDURE ModifyDBStruc(VAR alias: DBFile);
```

```
PROCEDURE Say(row, col: CARDINAL; s: ARRAY OF CHAR);
```

```
PROCEDURE Get(row, col: CARDINAL; VAR s: ARRAY OF
    CHAR);
```

```
PROCEDURE ReadGets;
```

```
PROCEDURE Row(): CARDINAL;
```

```
PROCEDURE Col(): CARDINAL;
```

```
END DBScreen.
```

DBScreen uses Repertoire's SmartScreen and WindowPrims modules to implement a simple form of input handling that resembles the one used by dBaseIII. Because the procedures are implemented through Repertoire's
© PMI 1993 All rights reserved.

screen system, however, a program that uses them has unusual flexibility. See Chapter 6 of the Repertoire manual for details.

EXAMPLE:

```
...
Say(5,5, 'First name: ');
Get(Row(), Col(), fname);
Say(6,5, 'Last name: ');
Set(Row(), Col(), lname);
ReadGets;
...
```

Say writes the string 's' to the screen beginning at the location specified by 'row' and 'col'. By default, the colors and monochrome attributes it uses are as follows: on monochrome monitors, input fields will appear in Reverse Video and normal text will appear as plain black on white; on color monitors, input fields are blue on light gray, and normal text is light gray on blue. It insulates calling procedures from its color choices by using Repertoire's PushColors and PopColors routines. Note, however, that it does not push and pop the cursor size or position, and does not turn it off, because we don't want the cursor to flash between fields as they are initially written. That means you ought to turn the cursor off (using Repertoire's SetCursorHeight procedure) before doing a series of Say and Get statements.

Get highlights a portion of the screen of the appropriate length and accepts input in the highlighted region. It uses Repertoire's new ReadWithEdits procedure to do so, thus allowing full editing functions in the string, including an insert/overwrite mode toggle that is controlled by pressing the Ins key.

ReadGets processes the information entered in the input areas of each variable specified by a Get command. The cursor keys may be used to move from input area to input area. ReadGets leaves the cursor at the bottom of the screen, as does READ in dBase III.

The variable 'lastdirection' is set to indicate in which direction the exit from the input screen was made. If the cursor is in the last input area and the return key is hit

'lastdirection' is set to Ahead. If the cursor is moved out of the last input area with the right or down arrow keys 'lastdirection' is set to Ahead. If the cursor is moved out of the first input area with the up or left arrow

keys 'lastdirection' is set to Back. If the Esc key is hit at anytime during ReadGets screen input ends and 'lastdirection' is set to Escape. During screen input 'lastdirection' is set to Nowhere except upon moving from field to field: then it reflects the direction of the move. Row() returns the row coordinate of the current cursor position. Col() returns the column coordinate of the current cursor position.

CreateDBF is an interactive, full-screen file creation procedure loosely based on the dBase CREATE command. It is provided primarily as an example of how your programs might handle the creation of files; you would probably want to improve its user interface before incorporating it into a finished program. It displays a simple heading that prompts for the name, type, size, and number of decimal places associated with each field. It uses the Say and Get routines in DBScreen to obtain new field definition data from the user, and puts that data into an ARRAY OF DBFieldDescriptor that can conveniently be passed to the BuildDBF procedure.

Locks

DEFINITION MODULE Locks;

```
(*
*
*)
```

```
* ModBase
* Release 3.0
* (c) Copyright 1991 John McMonagle
* P.O. Box 8402
* Green Bay WI 54308
* All Rights Reserved
```

```
FROM StringIO IMPORT ErrorMessage;
FROM ErrorManager IMPORT AConfirmProc;
IMPORT SYSTEM;
```

```
TYPE
    RangeRec =
        RECORD
            FileOffset : LONGINT;
            RangeLength: LONGINT;
        END;
    Range = POINTER TO RangeRec;
```

```
VAR
    AbortRetryIgnore:AConfirmProc;
    ExclusiveOnly:BOOLEAN;
```

```
PROCEDURE Lock(FileHandle : CARDINAL; range:RangeRec)
    : ErrorMessage;
```

```
PROCEDURE LockRetry(FileHandle : CARDINAL;
    range:RangeRec;
    Retrys:CARDINAL; VAR FileName:ARRAY OF CHAR) :
    ErrorMessage;
```

```
PROCEDURE UnLock(FileHandle : CARDINAL;
    range:RangeRec) : ErrorMessage;
```

```
PROCEDURE LockFile(FileHandle : CARDINAL) :
    ErrorMessage;
```

```
PROCEDURE LockFileRetry(FileHandle : CARDINAL;
    Retrys:CARDINAL; VAR FileName:ARRAY OF
    CHAR) : ErrorMessage;
```

```
PROCEDURE UnLockFile(FileHandle : CARDINAL) :
    ErrorMessage;
```

```

(* used to pause between retries should give give
   control to the operating system if
   possible *)
PROCEDURE Pause(Msec:CARDINAL);

PROCEDURE NoLocking(handle:CARDINAL):BOOLEAN;

PROCEDURE ReadRetry(FileHandle : CARDINAL; ptr :
    SYSTEM.ADDRESS; size,Retrys :
    CARDINAL;VAR FileName:ARRAY OF CHAR) : CARDINAL;

END Locks.

```

```

Range =POINTER TO RangeRec;

```

```

VAR
AbortRetryIgnore:AConfirmProc;
ExclusiveOnly:BOOLEAN;

```

Locks contains low-level procedures for locking and unlocking files and portions of files.

Range is a type for specifying part of a file to lock or unlock. AbortRetryIgnore is a procedure to handle file locking errors. It is assigned to simple procedure that asks the user if to Abort, Retry or Ignore. Exclusive if TRUE is used by ModBase to set all access to exclusive. Locks tests to see if it can lock a file in the initialization. If it can not it sets Exclusive to TRUE. You can set it to TRUE to speed access, or to restrict access on a network.

```

PROCEDURE Lock(FileHandle : CARDINAL; range:RangeRec)
    : ErrorMessage;

PROCEDURE LockRetry(FileHandle : CARDINAL;
    Range:RangeRec; Retrys:CARDINAL; VAR
    FileName:ARRAY OF CHAR) : ErrorMessage;

PROCEDURE UnLock(FileHandle : CARDINAL;
    range:RangeRec) : ErrorMessage;

PROCEDURE LockFile(FileHandle : CARDINAL) :
    ErrorMessage;

PROCEDURE LockFileRetry(FileHandle : CARDINAL;
    Retrys:CARDINAL; VAR FileName:ARRAY OF
    CHAR) : ErrorMessage;

PROCEDURE UnLockFile(FileHandle : CARDINAL) :
    ErrorMessage;

```


The preceding locks or unlocks files or portions of files. The 'Retry' variations retry Retrys number of times before calling the AbortRetryIgnore Procedure. The ErrorMessage is a standard DOS or OS/2 error message. Likely possibilities are 0 for no error, 5 for access denied, 32 for sharing violation.

```
PROCEDURE Pause(Msec:CARDINAL);
```

Pause is used to pause between retries and gives control to the operating system if possible.

```
PROCEDURE NoLocking(handle:CARDINAL):BOOLEAN;
```

NoLocking Checks if Locking is possible for a file handle

```
PROCEDURE ReadRetry(FileHandle : CARDINAL; ptr :  
SYSTEM.ADDRESS; size,Retrys :  
CARDINAL;VAR FileName:ARRAY OF CHAR) :  
CARDINAL;
```

Tries to read the file "Retrys" number of times before calling the AbortRetryIgnore procedure.

MemoFunctions

```
DEFINITION MODULE MemoFunctions;
```

```
( *
 *   ModBase
 *   Release 3.0
 *   By Don Fletcher & John McMonagle
 *   (c) Copyright 1986 - 1991 PMI
 *   P.O. Box 8402
 *   Green Bay WI 54308
 *   All Rights Reserved
 *
 *)
```

```
FROM ModBase3 IMPORT DBFile;
```

```
PROCEDURE GetMemoField(VAR alias: DBFile; fieldnum:
    CARDINAL; VAR memo: ARRAY OF CHAR);
```

```
PROCEDURE ReplaceM(VAR alias: DBFile; fieldnum:
    CARDINAL; VAR memo: ARRAY OF CHAR);
```

```
END MemoFunctions.
```

MemoFunctions provides procedures to read and write memo fields specified in DBFiles. dBase's cumbersome memo field type is its only way of dealing with text of indeterminate length. The actual memo field of the data file is only ten characters long; its value indicates the position of the text in an associated textfile.

```
PROCEDURE GetMemoField(VAR alias: DBFile; fieldnum:
    CARDINAL; VAR memo: ARRAY OF CHAR);
```

GetMemoField's fieldnum parameter should be the number of a memo field associated with 'alias'. The DBFile must be open, but ModBase will open the associated memofile automatically and close it when closing the DBFile. Your only responsibility is to make sure that the variable memo is large enough to hold all the characters read from 'MemoFileHandle'.

GetMemoField assumes an extension of .dbt on memofiles. Since the memo string may be very large, you may want to dynamically allocate memo in the heap instead of using local variables to help prevent stack overflows. This is an example from MemoFunctions.ReplaceM. Most of the code has been removed. To see the rest of it, look at MemoFunctions.MOD:

ModBase™ 3.0

```
PROCEDURE ReplaceM(VAR alias: DBFile; fieldnum:
                  CARDINAL; VAR memo: ARRAY OF CHAR);
VAR
  FileError: ErrorMessage;
  ptrstr: ARRAY [0..9] OF CHAR;
  blocknum, blockpos: LONGINT;
  OldMemo : POINTER TO ARRAY[0..MaxMemoSize] OF CHAR;
  PROCEDURE StrEqual(VAR str1, str2: ARRAY OF
                    CHAR): BOOLEAN;
  VAR
    limit, count: CARDINAL;
  BEGIN
    ...
  END StrEqual;
  (* if it were not for using StrEqual instead of
    CompareStr and using VStorage for
    allocations, one could easily use 16K of
    stack just in calling this procedure, if
    str was a local variable of the calling
    procedure and str is of MaxMemoSize *)
BEGIN
  IF NOT alias.MemoOpen THEN
    PrintMessage(OpenFile(alias.MemoHandle
                        , alias.MemoName));
    alias.MemoOpen := TRUE;
  END;
  IF memo[0] # 0C THEN
    (* check to see if memo changed *)
    NEW(OldMemo);
    GetMemoField(alias, fieldnum, OldMemo^);
    IF NOT StrEqual(OldMemo^, memo) THEN
      END;
      DISPOSE(OldMemo);
    ELSE
      Replace(alias, fieldnum, '      ');
    END;
  END ReplaceM;
```

```
PROCEDURE ReplaceM(VAR alias: DBFile; fieldnum:
                  CARDINAL; VAR memo: ARRAY OF CHAR);
```

ReplaceM follows dBase III in writing any changes to the end of the memo file. The file space occupied by a previous version of the memo is simply abandoned and not available for future use. Thus, frequently edited memo files become very long. ReplaceM will compare the current value of the memo text with the new value and will return without doing anything else if these values are the same. This takes time but helps keep memo files from growing needlessly, particularly for applications that use the Scrn2DBF module.

If the safety field in the DBFile record passed to ReplaceM's first parameter is TRUE, ModBase will force changes to the disk. Although memos are in separate files they are considered to be logically a part of a DBFile.

MiscFunctions

DEFINITION MODULE MiscFunctions;

```
( *
 *  ModBase
 *  Release 3.0
 *  By Don Fletcher & John McMonagle
 *  (c) Copyright 1986 - 1991 PMI
 *  P.O. Box 8402
 *  Green Bay WI 54308
 *  All Rights Reserved
 *)

PROCEDURE Alph(c: CHAR): BOOLEAN;

PROCEDURE Digit(c: CHAR): BOOLEAN;

PROCEDURE AlphaNum(c: CHAR): BOOLEAN;

PROCEDURE Punc(c: CHAR): BOOLEAN;

PROCEDURE Printable(c: CHAR): BOOLEAN;

PROCEDURE IsLower(c: CHAR): BOOLEAN;

PROCEDURE IsUpper(c: CHAR): BOOLEAN;

PROCEDURE IsSpace(c: CHAR): BOOLEAN;

PROCEDURE Trim(VAR str: ARRAY OF CHAR);

PROCEDURE Upper(VAR str: ARRAY OF CHAR);

PROCEDURE FieldName(str: ARRAY OF CHAR): BOOLEAN;

PROCEDURE FieldNameChar(c: CHAR): BOOLEAN;

END MiscFunctions.
```

Alph returns TRUE if 'c' is an alphabetic character (i.e., is between 'a' and 'z' or 'A' and 'Z', inclusive). **Digit** returns TRUE if 'c' is a digit (i.e., is between '0' and '9', inclusive). **AlphaNum** returns TRUE if 'c' is a digit or alphabetic character. **Punc** returns TRUE if 'c' is a punctuation mark. Note, however, that it assumes that characters with ordinal values above 127 are punctuation marks. **Printable** returns TRUE if the ordinal value of 'c' is between 32 and 127, inclusive. **IsLower** returns TRUE if 'c' is a lowercase alphabetic character. **IsUpper** returns TRUE if 'c' is an uppercase alphabetic character. **IsSpace** returns TRUE if 'c' is a space, an EOL, a carriage return,

a line feed, or a tab. Trim strips trailing blanks from a string. Upper converts a string to uppercase. FieldName returns TRUE if 'str' is a legal dBase field name; that requires merely that the string have 10 or fewer characters, all alphabetic or underscores. FieldNameChar returns TRUE if 'c' is an alphabetic character or an underscore.

ModBase3

DEFINITION MODULE ModBase3;

```
( *
 *   ModBase
 *   Release 3.0
 *   By Don Fletcher & John McMonagle
 *   (c) Copyright 1986 - 1991 PMI
 *   P.O. Box 8402
 *   Green Bay WI 54308
 *   All Rights Reserved
 *
 *)
```

FROM SYSTEM IMPORT
ADDRESS;

CONST

```
MaxRecLength = 4000;
MaxField = 128;
MaxHeaderLen = 32*(MaxField+1)+2;
NameLen = 64;
MemoSize = 512;
ReadOnly = {};
ReadWrite = {1};
Exclusive = {4};
PermitAll = {6};
PermitRead = {5};
```

TYPE

```
RecordModeType = (CurrentRec, Buffer, ReRead);
DBFieldDescriptor =
  RECORD
    name: ARRAY [0..9] OF CHAR;
    size: CARDINAL;
    decplaces: CARDINAL;
    offset: CARDINAL;
    fldtype, align: CHAR; (* align is to make size even
                          *)
  END;
```

```
DBFieldArray = ARRAY [1..MaxField] OF
  DBFieldDescriptor;
```

```
DBFieldPtr = POINTER TO DBFieldArray;
```

```
(* consider all of DBFile type read only *)
DBFile;
FixUpProcedure = PROCEDURE (DBFile): BOOLEAN;
```

```

IndexCheckProcedure=PROCEDURE( DBFile);

VAR
UpDateIndexes:IndexCheckProcedure;

PROCEDURE InitDBF(filename: ARRAY OF CHAR; VAR alias:
    DBFile; BufferSize:CARDINAL;
        safety,Exclusive,AutoLock:BOOLEAN;
    FixUp:FixUpProcedure );

PROCEDURE DisposeDBF(VAR alias:DBFile);

PROCEDURE OpenDBF
    (    alias    : DBFile ):BOOLEAN;

PROCEDURE BuildDBF( fields: ARRAY OF
    DBFieldDescriptor;NumFields:CARDINAL; alias:
    DBFile):CARDINAL;

PROCEDURE ClosedBF( alias: DBFile);

PROCEDURE UpdatedBFile( alias: DBFile);
    (* Used when safety is off to update DBFile and
    Memos to disk *)

PROCEDURE ReadDBRec( alias: DBFile; recnum: LONGINT) ;

PROCEDURE SetDBBuffer( alias: DBFile; BufferSize:
    CARDINAL );
    (* BufferSize in Bytes, Will set to multiple of
    alias.length *)

PROCEDURE WriteDBRec( alias: DBFile);

PROCEDURE GetField( alias: DBFile; fieldnumber:
    CARDINAL; VAR field: ARRAY OF CHAR);

PROCEDURE Replace( alias: DBFile; fieldnumber:
    CARDINAL;
    field: ARRAY OF CHAR);

PROCEDURE PosOfField( alias: DBFile;VAR fieldname:
    ARRAY OF CHAR): CARDINAL;

PROCEDURE AppendBlank( alias: DBFile);

PROCEDURE DeleteRecord( alias: DBFile);

PROCEDURE UnDeleteRecord( alias: DBFile);

PROCEDURE Deleted( alias: DBFile): BOOLEAN;

PROCEDURE SetDBSafetyOn( alias: DBFile);

PROCEDURE SetDBSafetyOff( alias: DBFile);

```


ModBase™ 3.0

```
PROCEDURE SetRecordMode( alias:
    DBFile;Mode:RecordModeType);

PROCEDURE LockRec( alias:
    DBFile;recnum:LONGINT):CARDINAL;

PROCEDURE UnLockRec( alias:
    DBFile;recnum:LONGINT):CARDINAL;

PROCEDURE DBError( alias:DBFile):CARDINAL;

PROCEDURE Appending(alias:DBFile):BOOLEAN;

PROCEDURE FileName(alias:DBFile;VAR Name:ARRAY OF
    CHAR);

PROCEDURE HasMemo(alias:DBFile):BOOLEAN;

PROCEDURE SafetySet(alias:DBFile):BOOLEAN;

PROCEDURE IndexList(alias:DBFile):ADDRESS;

PROCEDURE SetIndexList(alias:DBFile;ndx:ADDRESS);

PROCEDURE Record(alias:DBFile):LONGINT;

PROCEDURE FieldList(alias:DBFile):DBFieldPtr;

PROCEDURE BufferSize(alias:DBFile):CARDINAL;

PROCEDURE NumberOfFields(alias:DBFile):CARDINAL;

PROCEDURE NumberRecords(alias:DBFile):LONGINT;

PROCEDURE RecordPtr(alias:DBFile):ADDRESS;

PROCEDURE RecordLength(alias:DBFile):CARDINAL;

PROCEDURE OpenMemo(alias:DBFile):BOOLEAN;

PROCEDURE MemoHandle(alias:DBFile):CARDINAL;

PROCEDURE NilDBF(VAR alias:DBFile);

PROCEDURE Initalized(alias:DBFile):BOOLEAN;

PROCEDURE DefaultFixUp(alias:DBFile):BOOLEAN;

END ModBase3.
```

As of ModBase's 3.0 release, the process of opening an DBFile has been split into two steps. You must first initialize a DBFile variable with

InitDBFile. You then open the index by calling a separate OpenDBFile procedure.

ModBase3 exports a record type called DBFile that contains information about the structure of the dBase file. It also handles bookkeeping operations on open data files, and permits a program to get information about the file quickly. All exported procedures take a variable of type DBFile.

Note that since release 2.1, it is no longer safe to assume that the first unused field in the fieldlist (a POINTER TO DBFieldArray) will have a blank name. The reason is that ModBase now conserves memory by allocating only as much memory for the fieldlist as it actually needs. Therefore, if you reference elements of the array beyond numberoffields, you will see uninitialized memory. Use the procedure NumberOfFields to determine the size of the DBFieldArray.

In the order of their declaration in the definition module, this is the function of the more important identifiers in ModBase3:

```
UpDateIndexes: IndexCheckProcedure;
```

UpDateIndexes is a procedure variable that is used internally to update indexes that are changed by changes to a DBFile. It is assigned to a procedure in DBIndexes by DBindexs.

```
PROCEDURE InitDBF(filename: ARRAY OF CHAR; VAR alias:
    DBFile; BufferSize: CARDINAL;
    safety, Exclusive, AutoLock: BOOLEAN;
    FixUp: FixUpProcedure );
```

```
PROCEDURE DisposeDBF(VAR alias: DBFile);
```

```
PROCEDURE OpenDBF
    (    alias      : DBFile    ): BOOLEAN;
```

```
PROCEDURE CloseDBF( alias: DBFile);
```

In ModBase's the process of opening a data file has been split into two steps. First you must use the InitDBF procedure to initialize a variable of type DBFile. Then you may physically open the file and associate it with that DBFile variable by calling OpenDBF. When you are finished with a data file, you should call CloseDBF to close the DOS handle associated with the file and call.

InitDBF's BufferSize parameter must be specified in number of bytes. A minimum of 1 record length will be used, so if you want a small buffer, 0 is fine. In general, if you are reading randomly or appending records, a small buffer is best, and a large buffer is best if reading sequentially. The parameter safety if TRUE forces any changes to be written to the disk immediately. It is not needed but considerably reduces the chance of losing data in the case of a power failure of someone switching the computer off. Exclusive opens the DBFile with exclusive access that is no other processes will be able to access the file. If Exclusive is set and another process is using the DBFile the OpenDBF operation will fail. AutoLock if TRUE will use ModBase auto locking. That is files are only locked during WriteDBF operations. At that time the record is read and if it is currently the same as when previously read. The new values are written to the file. If the record has been changed the procedure passed as FixUp is called. The procedure DefaultFixUp if used will merge the changes on a field by field basis. If FixUp returns TRUE the record is to be written. You can provide your own procedure to do what ever you want, such as ask the user to verify, and/or manually intervened. What this does is reduces the amount of time records are locked to the minimum, while still catching simultaneous changes by separate processes. This is adequate in most database applications.

OpenDBF opens a disk file specified by the name associated with the DBFile in the filename parameter when you called InitDBF. It then checks to see if it is a dBaseIII type file, reads its header information, and assigns appropriate values to the variables in DBFile. The first record of the data file is read. Record will return 1 if there is at least 1 record 0 if there are none. If the operation is unsuccessful for any reason, OpenDBF returns FALSE. Note that the filename need not use the '.DBF' filename extension required by dBase. CloseDBF closes an opened dBase file. Use DisposeDBF to release memory allocated by InitDBF and OpenDBF

```
PROCEDURE BuildDBF( fields: ARRAY OF
                    DBFieldDescriptor; NumFields: CARDINAL;
                    alias: DBFile): CARDINAL;
```

BuildDBF makes a dBase type data file. To give it the proper field definitions, you must first initialize an array of DBFieldDescriptor, then pass that array to BuildDBF's second parameter. BuildDBF must have Exclusive access and will return an error if another process is using the DBFile. The offset field of the descriptors is ignored, since the offsets are

calculated by the OpenDBF procedure whenever the data file is used. BuildDBF will silently overwrite any file with the same name as filename. Before calling it, you should check for existing files and issue an appropriate warning. Note also that BuildDBF does not close the created file. Use CloseDBF for that purpose.

```
PROCEDURE ReadDBRec(alias: DBFile;
recnum:
LONGINT) ;
PROCEDURE WriteDBRec(alias: DBFile);
```

ReadDBRec reads the "recnum-th" record and stores it internally in the DBFile. If the file identified by alias is not open, or if recnum exceeds the number of records (i.e., the number of records in the file) it the read will fail. WriteDBRec writes the current record to disk. No changes made to the current record will be written to the data file unless WriteDBRec is called. A feature inherited from dBase is that the first character in the current record is an asterisk (*) if the record is deleted.

Your changes to a record do not take effect unless you write to the disk, including changes to memos.

```
PROCEDURE DeleteRecord (alias: DBFile);
PROCEDURE UnDeleteRecord( alias: DBFile);
PROCEDURE Deleted(alias: DBFile): BOOLEAN;
```

The procedure DeleteRecord marks the current record for deletion and writes the change to the file. Deleted records may be returned to the database by UnDeleteRecord, unless some variant of the dBase PACK operation has been performed on them. It writes a blank in the first byte of the current record and writes the change to the file. UnDeleteRecord sets the current record of alias to normal status record. The function Deleted() returns TRUE if the current record has been marked for deletion.

```
PROCEDURE GetField(alias: DBFile; fieldnumber:
CARDINAL; VAR field: ARRAY OF CHAR);

PROCEDURE Replace( alias: DBFile; fieldnumber:
CARDINAL; field: ARRAY OF CHAR);
```

Individual fields of the current record (i.e., the record last read from the DBFile identified first in GetField's parameter list) can be extracted by the procedure GetField. It retrieves the value of the field specified by fieldnumber and copies it into the string passed to 'field'. Extra characters

© PMI 1993 All rights reserved. 41

are dropped if 'field' is not large enough to accommodate the string read from the data file. GetField does not look for a field by name, but by number. That makes it much faster, since no string matching is required. The procedure GetNamedField in DBList allows you to locate a field by name. GetField is the main routine for getting data from a record. All data is in ASCII form. The DBFields module has additional routines for getting and changing fields. Replace does just the opposite of GetField. You should use procedures in DBFields for fields that are not character fields.

```
PROCEDURE PosOfField(alias: DBFile; fieldname:ARRAY OF  
CHAR): CARDINAL;
```

PosOfField is useful for finding the position of a DBase3 field. Currently all ModBase fields are referenced by position, which can sometimes cause problems if you add or delete fields. A good solution is to allocate variables for all the field numbers you are using and use PosOfField to get all the field positions once.

```
PROCEDURE AppendBlank(alias: DBFile);
```

AppendBlank is the procedure that adds new records to the data file. New in Version 3 AppendBlank does not actually add a record but logically places you at the end of the file. This is much faster and makes it easier to abort from an Append operation. After AppendBlank the current record number is undefined. If you need a record number before you write it. Do a WriteDBF, then Record will return the current record number.

```
PROCEDURE SetDBBuffer( alias: DBFile; BufferSize:  
CARDINAL );
```

Used to change the buffer size BufferSize in Bytes, Will set to multiple of the length of a record.

```
PROCEDURE UpdateDBFile(alias: DBFile);
```

UpdateDBFile has the effect of closing and opening the DBFile but is faster. It is used when safety is off to update DBFile and Memos to disk. Safety has a significant speed penalty on writes, but Safety will update the drive immediately, including the header information in the case of append. Meaning that data will not be lost from unexpected termination (like power loss, etc.) as long as it can complete the disk writes. In general use safety on during data entry, safety off if doing a long sequence of appends. It is a good Idea to Call UpdateDBFile when changing from Safety off to on. It would be a good idea to use termination procedures to close DBFiles,

especially if you normally have safety off. For you will likely loose data if CloseDBF is not called at termination, for although DOS may close your open files the beginning of the file contains information about how many records a file has and this is only updated when the file is closed.

```
PROCEDURE SetDBSafetyOn( alias: DBFile);
```

```
PROCEDURE SetDBSafetyOff( alias: DBFile);
```

SetDBSafetyOn and SetDBSafetyOff set the safety state for a DBFile.

```
PROCEDURE SetRecordMode( alias: DBFile;  
                        Mode:RecordModeType);
```

SetRecordMode sets which buffer is used by GetField and Replace operations. You should not need to use this.

```
PROCEDURE LockRec( alias: DBFile;  
                  recnum:LONGINT):CARDINAL;
```

```
PROCEDURE UnLockRec( alias: DBFile;  
                    recnum:LONGINT):CARDINAL;
```

LockRec and UnLockRec are used to manually lock and unlock records. If you are using exclusive access or automatic locking you should not need to use these procedures. Remember that a locked record can not be accessed by another process period.

```
PROCEDURE DBError( alias:DBFile):CARDINAL;
```

DBError returns the last file error code from the last operation. If using the automatic locking mode you will seldom have to concern yourself with these codes. If locking manually you should check for errors afire the following operations: OpenDBFile, ReadDBRec, WriteDBRec, DeleteRecord and UnDeleteRecord.

```
PROCEDURE Appending(alias:DBFile):BOOLEAN;
```

Indicates if the DBFile is an appending state, and the file is logically at EOF, and the current record number is undefined.

```
PROCEDURE FileName(alias:DBFile;VAR Name:ARRAY OF CHAR);
```

Returns the Name of the DBFile.

```
PROCEDURE HasMemo(alias:DBFile):BOOLEAN;
```


ModBase™ 3.0

HasMemo Indicates if the DBFile has a Memo file.

```
PROCEDURE SafetySet(alias:DBFile):BOOLEAN;
```

Indicates if safety is set for the current DBFile.

```
PROCEDURE IndexList(alias:DBFile):ADDRESS;
```

IndexList points to the list of associated Indexes. It is used internally.

```
PROCEDURE SetIndexList(alias:DBFile;ndx:ADDRESS);
```

Is used internally to manipulate the list of associated indexes.

```
PROCEDURE Record(alias:DBFile):LONGINT;
```

Record returns the current record number. If appending it returns MAX(LONGINT).

```
PROCEDURE FieldList(alias:DBFile):DBFieldPtr;
```

Returns a pointer to the fields for the DBFile.

```
PROCEDURE BufferSize(alias:DBFile):CARDINAL;
```

Returns the size of the current read buffer in bytes.

```
PROCEDURE NumberOfFields(alias:DBFile):CARDINAL;
```

Returns the number of fields in the DBFile.

```
PROCEDURE NumberRecords(alias:DBFile):LONGINT;
```

Returns the number of records in the current DBFile.

```
PROCEDURE RecordPtr(alias:DBFile):ADDRESS;
```

Gives a pointer to the current record.

```
PROCEDURE RecordLength(alias:DBFile):CARDINAL;
```

Returns the length of records in DBFile.

```
PROCEDURE OpenMemo(alias:DBFile):BOOLEAN;
```

Open the Memo associated with the current DBFile. There is no need for you to ever do this as it is done by MemoFunctions.

```
PROCEDURE MemoHandle(alias:DBFile):CARDINAL;
```

Returns the file handle of the Memo file. You should never need to use this as memo files are treated logically as part of the DBFile.

```
PROCEDURE NilDBF(VAR alias:DBFile);
```

Sets the DBFile to NIL. This is not normally needed.

```
PROCEDURE Initalized(alias:DBFile):BOOLEAN;
```

Tell whether a DBFile has been Initialized.

```
PROCEDURE DefaultFixUp(alias:DBFile):BOOLEAN;
```

DefaultFixUp is a FixUpProcedure you may use. It is called in WriteDBFile if it finds that the current record has been changed since the last time it was read. This procedure merges the changes on a field by field bases.

Scrn2DBF

DEFINITION MODULE Scrn2DBF;

```
( *
 *  ModBase
 *  Release 3.0
 *  By Don Fletcher & John McMonagle
 *  P.O. Box 8402
 *  Green Bay WI 54308
 *  All Rights Reserved
 *  (c) Copyright 1988 - 1991 PMI
 *
 *)
```

FROM ModBase3 IMPORT DBFile;

FROM ScrnTypes IMPORT DisplayFrame;

```
PROCEDURE FrameToDBF(VAR DBF: DBFile; VAR TheFrame:
  DisplayFrame): BOOLEAN;
```

```
PROCEDURE DBFToFrame(VAR DBF: DBFile; VAR TheFrame:
  DisplayFrame);
```

END Scrn2DBF.

Scrn2DBF provides routines that perform automatic transfers of data between ModBase's data files and Repertoire's screen system. It serves the same purpose for ModBase that Repertoire's Scrn2Ndx module serves for its own NdxFile system. For explanation of the Repertoire screen system and a full explanation of how data is transferred into and out of it, please see the Repertoire manual.

The general idea is that these routines transfer data between fields that have the same name in both the DisplayFrame and the DBFile. You can have unshared field names in both the frame and the DBFile; only the shared field names will be affected. Note that the matching process requires the use of UPPERCASE field names in the DBFile. Repertoire's screen system always stores field names in upper case (and in fact, so does ModBase's DBCreate.CreateDBF procedure). Since it is possible to create a DBFile with lowercase field names, however, you should keep this in mind if you use some other procedure to create your files.

In keeping with ModBase's generally transparent handling of memo files, Scrn2DBF automatically transfers memo fields into Repertoire editor

fields, and vice versa. (Editor fields are ideal for handling variable-length text data because text in an editor field wrap and scrolls horizontally and vertically in a little window that the user perceives simply as a normal data entry field.)

```
PROCEDURE FrameToDBF(VAR DBF: DBFile; VAR TheFrame:
    DisplayFrame): BOOLEAN;
```

FrameToDBF copies the fields from DisplayFrame To DBFile that have the same name. FrameToDBF still has a Boolean result for purposes of compatibility with earlier releases, even though the result is always TRUE. The function result was used in validating Date fields, but that is no longer necessary because versions of Repertoire following 1.5c check Date fields in the ControlFrame procedure.

```
PROCEDURE DBFToFrame(VAR DBF: DBFile; VAR TheFrame:
    DisplayFrame);
```

DBFToFrame copies fields from DBFile to DisplayFrame that have the same name.

Index

A

AbortRetryIgnore, 28
 Adding a Record, 42
 AddRecord, 16, 20
 AddToUpdateList, 7, 16, 22
 Alph, 34

AlphaNum, 34

AppendBlank, 37, 42
 Appending, 38, 43

B

BufferSize, 38, 44
 BuildCompIndex, 16, 22
 BuildDBF, 37, 40
 BuildIndex, 16, 21

C

ChkInd, 9
 CloseDBF, 8, 37, 39
 CloseIndex, 8, 16, 18
 Col(), 25
 Complex index procedure
 example, 21
 CreateDBF, 25, 27
 CurrentKeyCh, 16, 20
 CurrentKeyN, 16, 20
 CurrentRec, 16, 20

D

Database buffer size, 40
 Database creation, 40
 DataFormatToDate, 12, 13
 DateToDataFormat, 12, 13
 DBCopier, 10
 DBCopy, 10
 DBError, 38, 43
 DBFieldArray, 36
 DBFieldCopy, 10, 11
 DBFieldDescriptor, 36
 DBFieldPtr, 36

DBFields, 6, 12

DBFile, 36

DBFToFrame, 46, 47

DBIndexes, 6, 15

DBIndexes.FindPositionCh, 21

DBIndexes.InsertEntry, 21

DBPack, 10

DBRelate, 24

DBScreen, 25

DefaultFixUp, 38, 45

DefaultKeyProcedure, 15, 18

DeleteCurrentEntry, 16, 22

Deleted, 37, 41

DeleteRecord, 37, 41

Deleting a record, 41

Digit, 34

DisposeDBF, 37, 39

DisposeIndex, 15, 19

E

EnterLock, 17, 23

ErrorManager, 6

Example of auto-locking, 6

Exclusive access setting, 29

ExclusiveOnly, 28

ExitLock, 17, 23

F

FieldList, 38, 44

FieldName, 34

FieldNameChar, 34

FileName, 38, 43

FindPositionCh, 15, 18

FindPositionN, 15, 18

FixUpProcedure, 36

FrameToDBF, 46, 47

G

Get, 25

GetDateField, 12, 13

GetDescriptors, 25
 GetField, 37, 41
 GetLogicalField, 12, 13
 GetMemoField, 31
 GetNumField, 12
 GoBottom, 16, 20
 GoTop, 16, 20
 H
 HasMemo, 38
 I
 Index maintaining automatically, 22
 Index buffers, 19
 Index example of record read, 18
 Index key in caps, 22
 Index node size, 19
 Index sequential access, 19
 IndexCheckProcedure, 37
 Indexes with unique keys, 23
 IndexList, 38, 44
 Initalized, 38, 45
 InitCompilerMods, 6
 InitCompIndex, 15, 18, 21
 InitDBF, 7, 37, 39
 InitIndex, 7, 15, 18
 InsertEntry, 15, 21
 IsLower, 34
 IsSpace, 34
 IsUpper, 34
 K
 KeyLength, 16, 20
 KeyProcedure, 15, 18
 L
 Lock, 28, 29
 LockFile, 28, 29
 LockFileRetry, 28, 29
 LockRec, 38, 43
 LockRetry, 28, 29
 Locks, 6, 28
 Locks.Pause, 7
 © PMI 1993 All rights reserved.

M
 MaxField, 36
 MaxRecLength, 36
 MemoFunctions, 31
 MemoHandle, 38, 44
 MemoSize, 36
 MiscFunctions, 34
 ModBase, 3
 ModBase features, 4
 ModBase3, 6, 36
 ModBase3.GetField, 21
 ModifyDBStruc, 25
 N
 NDXChk, 9
 NextRecord, 16, 19
 NilDBF, 38, 45
 NoLocking, 29, 30
 NumberOfFields, 38, 44
 NumberRecords, 38, 44
 Numeric keys, 18
 NumKeyType, 16, 20
 O
 OpenDBF, 7, 37, 39, 40
 OpenIndex, 15, 18
 OpenMemo, 38, 44
 P
 Pause, 29, 30
 PosOffField, 37, 42
 PrevRecord, 16, 19
 Printable, 34
 Punc, 34
 R
 RangeRec, 28
 ReadDBRec, 7, 37, 41
 ReadGets, 25
 Reading a database record, 41
 ReadRelate, 24
 ReadRetry, 29, 30
 Record, 38, 44

RecordLength, 38, 44
 RecordModeType, 36
 RecordPtr, 38, 44
 Repertoire, 46
 Replace, 8, 12, 13, 37, 41
 ReplaceD, 12, 13
 ReplaceL, 12, 13
 ReplaceM, 31, 32
 ReplaceN, 12, 13
 Row(), 25
 S

SafetySet, 38, 44
 Say, 25
 Scrn2DBF, 46
 Scrn2DBF.FrameToDBF, 21
 SetDBBuffer, 37, 42
 SetDBSafetyOff, 37, 43
 SetDBSafetyOn, 37, 43
 SetIndexBuffers, 16
 SetIndexList, 38, 44
 SetRecordMode, 38, 43
 SetSafetyOff, 16
 SetSafetyOn, 16
 T

Terms of Sale, 3
 Timing, 6
 Trim, 34

U

UnDeleteRecord, 37, 41
 UnLock, 28, 29
 UnLockFile, 28, 29
 UnLockRec, 38, 43
 UpdateDBFile, 37, 42
 UpdateDBIndexes, 16
 UpdateIndex, 16, 22
 UpdateIndexes, 37, 39
 UpdateUnique, 16, 23
 Upper, 34
 W

WriteDBRec, 8, 37, 41

RecordLength 35, 44
RecordMaxType 36
RecordPos 35, 41
RepairDB 46
Replace 8, 12, 13, 37, 41
ReplaceD 12, 13
ReplaceI 12, 13
ReplaceM 31, 32
ReplaceN 12, 13
Reset 35
S
SafetySet 35, 44
Say 28
Scan2DBF 46
Scan2DBF FrameToDBF 21
SetDBBuffer 42
SetDBSafetyOff 37, 43
SetDBSafetyOn 37, 43
SetIndexBuffer 16
SetIndexList 35, 44
SetIndexMode 35, 43
SetSafetyOff 43
SetSafetyOn 43
T
Terms of Sale 3
Timing 6
Trim 34
U
UnblockRecord 37, 41
Unlock 28, 29
UnlockFile 28, 29
UnlockRec 35, 43
UpdateDBFile 37, 42
UpdateDIndex 16
UpdateIndex 16, 22
UpdateIndexes 37, 39
UpdateUnique 16, 23
Upper 34
W

Symbol 32, 33, 41

