

ModBase

POLYTRON

High Quality Software Since 1982

ModBase

Release 2.1

by

Donald Fletcher
John McMonagle

from

POLYTRON Corporation

(c) Copyright 1986-1989 POLYTRON Corp.
All Rights Reserved

ModBase is a trademark of Donald G. Fletcher, John McMonagle and POLYTRON Corporation.
Repertoire is a registered trademark of POLYTRON Corporation.

Table of Contents

Terms of Sale.....	4
1.0 ModBase.....	5
1.1 Compiler Interface.....	5
1.2 Overview and Examples.....	7
1.3 ChkInd.....	10
1.4 DBCopier.....	11
1.5 DBCreate.....	13
1.6 DBFields.....	15
1.7 DBIndexes.....	18
1.8 DBList.....	26
1.9 DBRelate.....	28
1.10 DBScreen.....	30
1.11 MemoFunctions.....	33
1.12 MiscFunctions.....	36
1.13 ModBase3.....	38
1.14 Scrn2DBF.....	45
1.15 Definition Module Compilation Order.....	47
Index.....	48

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 Oregon shall have exclusive jurisdiction over any lawsuit concerning ModBase; you agree that Portland, Oregon, 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, 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.

1.0 ModBase

ModBase provides programmers with the tools to access data files created by the industry standard database manager, Ashton-Tate's dBase III. Although the procedures in the ModBase3 and DBCreate modules produce data files compatible with dBase, they can be used independently. Accordingly, ModBase includes accessory modules to handle B-Tree file indexing and manipulation.

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.

1.1 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 MODBASE.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).

If you are using a language other than C or Modula-2, you will have to translate the C header files into your own language. It is easier to work from the C header files than from the Modula-2 definition modules because they make clear precisely what is passed in each parameter list; an open-array parameter in Modula-2, for example, is actually two parameters in one.

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. The Modula-2 compiler initialization procedures are all named AssignProcs(), but are implemented differently for each compiler, and are stored in separate modules. The module intended for use with the Logitech compiler, for example, is named InitLogitech. Similar modules for other compilers are named InitJPI, InitStonyBrook, InitFST, etc. Please examine the ModBase distribution disks to find the Init[CompilerName] module for your compiler. 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.

ModBase was written by Donald Fletcher, of New York City, and John McMonagle, of Marinette, 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 is also a microcomputer consultant. He owns a company that specializes in software and services for the broadcast industry.

In C, you do much the same thing. First, include the following code to declare the function pointers that ModBase uses to access the C memory management routines:

```
#include "vstorage.h" #include <malloc.h> #include <stdio.h>

typedef void
(far * pascal AllocType) (void far *far *ptr, unsigned
    objectsz);

typedef int
(far * pascal AvailType) (unsigned amount);

int pascal AvailShell( amount )
    unsigned amount;
{
    char far *tmpptr;

    if (_freect(amount) > 0 )
        return(1);
    else
    {
        tmpptr = _fmalloc( amount );
        if (tmpptr == NULL)
            return(0);
        else
        {
            _ffree( tmpptr );
            return(1);
        }
    }
}

void pascal AllocShell( ptr, objectsz)
    void far * far*ptr;
    unsigned objectsz;
{
    *ptr = _fmalloc( objectsz );
}

void pascal DeallocShell( ptr, objectsz)
    void far *far *ptr;
    unsigned objectsz;
{
    _ffree( *ptr);
    *ptr = NULL;
}
```

You may want to turn this into an include file so that you can insert it into any program with a single line.

Next, include the .H files for all ModBase modules you want to use in your main function. For example:

```
#include "ModBase3.h"
```


On the first lines of main, declare any variables local to main, and then call the initialization routine for each ModBase module that main calls. A module's initialization routine is always a parameterless function named after the module itself. Finally, assign the function pointers declared above to the three VStorage procedure variables through which ModBase accesses your compiler's memory management services. This is a complete example of a C main function that declares a DBFile record, calls the initialization procedure for the ModBase3 module, initializes the VStorage procedure variables, and uses OpenDBF to open a data file named "customer.dbf":

```
main() {
    DBFile DbFileRec;

    ModBase3();

    VStorage_DosAlloc = (AllocType) AllocShell;
    VStorage_DosDealloc = (AllocType) DeallocShell;
    VStorage_DosAvail = (AvailType) AvailShell;

    ModBase3_InitDBF( 12, "customer.dbf", &DbFileRec, 0, TRUE );
    ModBase3_OpenDBF( &DbFileRec );
}
```

1.2 Overview and Examples

This is a table of procedures in each of ModBase's modules:

<u>ChkInd</u>	<u>DBScreen</u>
NDXChk	readattr
	writeattr
	MonoAttr
<u>DBCopier</u>	CreateDBF
	GetDescriptors
DBCoppy	ModifyDBStruc
DBPack	Say
DBFieldCopy	Get
	ReadGets
<u>DBCreate</u>	Row
	Col
BuildDBF	<u>MaxFiles</u>
<u>DBFields</u>	<u>MemoFunctions</u>
DateToDataFormat	
DataFormatToDate	GetMemoField
GetDateField	ReplaceM
GetLogicalField	
GetNumField	<u>MiscFunctions</u>
Replace	
Replaced	
ReplaceL	

Replacen

DBIndexes

FindPositionCh
FindPositionN
InitIndex
InitCompIndex
OpenIndex
DisposeIndex
InsertEntry
CloseIndex
UpdateIndex
SetSafetyOn
SetSafetyOff
SetIndexBuffers
NextRecord
PrevRecord
BuildIndex
BuildCompIndex
AddRecord
CurrentKeyCh
CurrentKeyN
CurrentRec
KeyType
KeyLength
DeleteCurrentEntry
GoTop
GoBottom
AddToUpdateList
UpdateUnique

DBList

List
GetNamedField

DBRelate

ReadRelate
PosOfField

Alph
Digit
AlphaNum
Punc
Printable
IsLower
IsUpper
IsSpace
Trim
Upper
FieldName
FieldNameChar

ModBase3

UpDateIndexes
InitDBF
OpenDBF
CloseDBF
UpdateDBFile
ReadDBRec
SetDBBuffer
WriteDBRec
GetField
AppendBlank
DeleteRecord
UnDeleteRecord
Deleted
SetDBSafetyOn
SetDBSafetyOff
SetCurrToDBBuffer
SetCurrToDBCur

Scrn2DBF

FrameToDBF
DBFToFrame

Not listed above are several Repertoire modules upon which ModBase depends for certain low-level purposes. ModBase.LIB contains these Repertoire object modules as well as the ones above, but they are meant only for internal use. You do not need to concern yourself with them.

Your distribution disks contain a simple example program named CheckRegister. It consists of CHECKREG.MOD, which contains the Modula-2 source code for the example; CHECKREG.REG, which is the data file that the program manipulates; and CHECKREG.NUM, which is an index for the data file. CheckRegister's source code demonstrates most of ModBase's facilities; it uses one data file and one index file to maintain a

simple check register.

The purpose of the CheckRegister is primarily to show you its source code. It isn't much to look at on screen; its user interface is built entirely from the simple routines in the DBScreen module. It requests a filename, to which it appends the extension '.REG'. If the file does not exist, it asks for confirmation before creating a new file. If it finds the file, it asks whether to edit existing entries or to add new entries. When editing, CheckRegister first asks for a transaction number. Thereafter, it proceeds either forward or backward in the file, depending on whether the user leaves the screen from the top or the bottom. If the user presses Esc, the program prompts for a new transaction number. The '.REG' extension must be changed to '.DBF' before dBase III will be able to use the file.

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 Scm2Dbf module, which transfers data automatically between Repertoire's screen system and a ModBase data file. Scm2Dbf is explained below. Please see Chapter 3 of the Repertoire manual for a complete explanation of Repertoire's screen system.

1.3 ChkInd

```
(*Start of JPI directives :*)
(*$D DATA*)
(*$M ModBase*)
(*End of JPI directives*)
```

```
DEFINITION MODULE ChkInd;
```

```
(*
*   ModBase
*   Release 2.1
*   By Don Fletcher & John McMonagle
*   (c) Copyright 1986, 1987, 1988 PMI
*   4536 SE 50th
*   Portland, Oregon 97206
*   All Rights Reserved
*)
```

```
FROM DBIndexes IMPORT
    FindPositionCh, OpenIndex, BuildIndex, DBIndex,
    NextRecord, PrevRecord;
```

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

```
END ChkInd.
```

C Header File:

```
#ifndef ChkInd_Def
#define ChkInd_Def

void far pascal ChkInd();

#include "DBIndexes.H"

extern void pascal far ChkInd_NDXChk( DBIndex far *ndx);
#endif
```

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.

1.4 DBCopier

```
(*Start of JPI directives :*)
(*$D DATA*)
(*$M ModBase*)
(*End of JPI directives*)
```

```
DEFINITION MODULE DBCopier;
```

```
( *
 *   ModBase
 *   Release 2.1
 *   By Don Fletcher & John McMonagle
 *   (c) Copyright 1986, 1987, 1988 PMI
 *   4536 SE 50th
 *   Portland, Oregon 97206
 *   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.
```

C Header File:

```
#ifndef DBCopier_Def
#define DBCopier_Def
```

```
void far pascal DBCopier();
```

```
#include "ModBase3.H"
```

```
extern void pascal far DBCopier_DBCopy( DBFile far *Source,
    DBFile far *Dest, int CopyDeleted);
```

```
extern void pascal far DBCopier_DBPack( DBFile far *dbf);
```

```
extern void pascal far DBCopier_DBFieldCopy( DBFile far *Source,
    DBFile far *Dest);
```

```
#endif
```



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

DBCopY 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; DBCopY 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. DBCopY'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, DBCopY 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.

1.5 DBCreate

```
(*Start of JPI directives :*)
(*$D DATA*)
(*$M ModBase*)
(*End of JPI directives*)
DEFINITION MODULE DBCreate;
(
*   ModBase
*   Release 2.1
*   By Don Fletcher & John McMonagle
*   (c) Copyright 1986, 1987, 1988 PMI
*   4536 SE 50th
*   Portland, Oregon  97206
*   All Rights Reserved
*
*)

FROM ModBase3 IMPORT DBFile, DBFieldDescriptor;

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

END DBCreate.
```

C Header File:

```
#ifndef DBCreate_Def
#define DBCreate_Def

void far pascal DBCreate();

#include "ModBase3.H"

extern void pascal far DBCreate_BuildDBF(
    unsigned High_filename, char far *filename,
    unsigned High_fields, DBFieldDescriptor far *fields,
    unsigned NumFields, DBFile far *alias);
#endif
```

DBCreate handles the creation of dBaseIII type data files. Since applications will often have to open and close data files, but need not create new files, the file creation procedures have their own module to save code space for applications that do not need them.

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

BuildDBF makes a dBase type data file. To give it the proper field definitions, you must first initialize an array of DBFieldDescriptors, then pass that array to BuildDBF's second

parameter. 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.

1.6 DBFields

```
(*Start of JPI directives :*)
(*$D DATA*)
(*$M ModBase*)
(*End of JPI directives*)
```

```
DEFINITION MODULE DBFields;
```

```
(*
 * ModBase
 * Release 2.1
 * By Don Fletcher & John McMonagle
 * (c) Copyright 1986, 1987, 1988 PMI
 * 4536 SE 50th
 * Portland, Oregon 97206
 * 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.
```

C Header File:


```

#ifndef DBFields_Def
#define DBFields_Def

void far pascal DBFields();

#include "ModBase3.H"
#include "DateFunctions.H"
#include "NumTypes.H"

extern void pascal far DBFields_DateToDataFormat( Date d,
    unsigned High_datestring, char far *datestring);

extern void pascal far DBFields_DataFormatToDate(
    unsigned High_dbstring, char far *dbstring, Date far *d);

extern void pascal far DBFields_GetDateField( DBFile far *alias,
    unsigned fieldnum, Date far *d);

extern void pascal far DBFields_GetLogicalField( DBFile far *alias,
    unsigned fieldnum, int far *l);

extern void pascal far DBFields_GetNumField( DBFile far *alias,
    unsigned fieldnum, Real8 far *n);

extern void pascal far DBFields_Replace( DBFile far *alias,
    unsigned fieldnum, unsigned High_s, char far *s);

extern void pascal far DBFields_Replaced( DBFile far *alias,
    unsigned fieldnum, Date d);

extern void pascal far DBFields_ReplaceL( DBFile far *alias,
    unsigned fieldnum, int l);

extern void pascal far DBFields_ReplaceN( DBFile far *alias,
    unsigned fieldnum, Real8 n);

#endif

```

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 alias.currentrec with a string representing that date. ReplaceL accepts a BOOLEAN variable and replaces the field specified by fieldnum in alias.currentrec with a string representing that value. ReplaceN accepts a REAL variable and replaces the field specified by fieldnum in alias.currentrec with a string representing that value.

1.7 DBIndexes

```
(*Start of JPI directives :*)
(*$D DATA*)
(*$M ModBase*)
(*End of JPI directives*)
DEFINITION MODULE DBIndexes;
```

```
(*
*   ModBase
*   Release 2.1
*   By Don Fletcher & John McMonagle
*   (c) Copyright 1986, 1987, 1988 PMI
*   4536 SE 50th
*   Portland, Oregon 97206
*   All Rights Reserved
*)
```

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

TYPE

DBIndex;

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

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

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

PROCEDURE InitIndex(indexname: ARRAY OF CHAR; VAR
ndx: DBIndex; buffersize: CARDINAL; safety:BOOLEAN);

PROCEDURE InitCompIndex(indexname: ARRAY OF CHAR; VAR
ndx: DBIndex; Key :KeyProcedure; buffersize: CARDINAL
; safety:BOOLEAN);

PROCEDURE OpenIndex(VAR ndx: DBIndex):BOOLEAN;

PROCEDURE DisposeIndex(VAR ndx: DBIndex);

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

PROCEDURE CloseIndex(VAR ndx: DBIndex);

PROCEDURE UpdateIndex(VAR ndx: DBIndex);

PROCEDURE SetSafetyOn(VAR ndx: DBIndex);

PROCEDURE SetSafetyOff(VAR ndx: DBIndex);


```

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

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

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

PROCEDURE BuildIndex(VAR ndx: DBIndex; VAR alias:
    DBFile; keyexp: CARDINAL);

PROCEDURE BuildCompIndex( VAR ndx: DBIndex; type: CHAR; VAR
    alias: DBFile; keyexp, size : CARDINAL);

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

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

PROCEDURE CurrentKeyN(VAR ndx: DBIndex): Real8;

PROCEDURE CurrentRec(VAR ndx: DBIndex): LONGINT;

PROCEDURE KeyType(VAR ndx: DBIndex): CHAR;

PROCEDURE KeyLength(VAR ndx: DBIndex): CARDINAL;

PROCEDURE DeleteCurrentEntry(VAR ndx: DBIndex);

PROCEDURE GoTop(VAR ndx: DBIndex);

PROCEDURE GoBottom(VAR ndx: DBIndex);

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

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

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

END DBIndexes.

```

C Header File:

```

#ifndef DBIndexes_Def
#define DBIndexes_Def

void far pascal DBIndexes();

#include "ModBase3.H"
#include "NumTypes.H"

```



```

typedef void far *DBIndex;

typedef void (far * pascal KeyProcedure) (DBFile far *,
    DBIndex far *, char far *);

extern void pascal far DBIndexes_FindPositionCh( DBIndex far *ndx,
    unsigned High_keystr, char far *keystr, int far *found);

extern void pascal far DBIndexes_FindPositionN( DBIndex far *ndx,
    unsigned High_keystr, char far *keystr, int far *found);

extern void pascal far DBIndexes_InitIndex(
    unsigned High_indexname, char far *indexname, DBIndex far *ndx,
    unsigned buffersize, int safety);

extern void pascal far DBIndexes_InitCompIndex(
    unsigned High_indexname, char far *indexname, DBIndex far *ndx,
    KeyProcedure Key, unsigned buffersize, int safety);

extern int pascal far DBIndexes_OpenIndex( DBIndex far *ndx);

extern void pascal far DBIndexes_DisposeIndex( DBIndex far *ndx);

extern void pascal far DBIndexes_InsertEntry( DBIndex far *ndx,
    unsigned High_kstr, char far *kstr, long recno);

extern void pascal far DBIndexes_CloseIndex( DBIndex far *ndx);

extern void pascal far DBIndexes_UpdateIndex( DBIndex far *ndx);

extern void pascal far DBIndexes_SetSafetyOn( DBIndex far *ndx);

extern void pascal far DBIndexes_SetSafetyOff( DBIndex far *ndx);

extern void pascal far DBIndexes_SetIndexBuffers( DBIndex far *ndx,
    unsigned Buffers);

extern int pascal far DBIndexes_NextRecord( DBIndex far *ndx,
    long far *recno);

extern int pascal far DBIndexes_PrevRecord( DBIndex far *ndx,
    long far *recno);

extern void pascal far DBIndexes_BuildIndex( DBIndex far *ndx,
    DBFile far *alias, unsigned keyexp);

extern void pascal far DBIndexes_BuildCompIndex( DBIndex far *ndx,
    char type, DBFile far *alias, unsigned keyexp, unsigned size);

extern void pascal far DBIndexes_AddRecord( DBFile far *alias,
    DBIndex far *ndx);

extern void pascal far DBIndexes_CurrentKeyCh( DBIndex far *ndx,
    unsigned High_val, char far *val);

```



```

extern Real8 pascal far DBIndexes_CurrentKeyN( DBIndex far *ndx);
extern long pascal far DBIndexes_CurrentRec( DBIndex far *ndx);
extern char pascal far DBIndexes_KeyType( DBIndex far *ndx);
extern unsigned pascal far DBIndexes_KeyLength( DBIndex far *ndx);
extern void pascal far DBIndexes_DeleteCurrentEntry( DBIndex far *ndx);
extern void pascal far DBIndexes_GoTop( DBIndex far *ndx);
extern void pascal far DBIndexes_GoBottom( DBIndex far *ndx);
extern void pascal far DBIndexes_AddToUpdateList( DBFile far *alias,
    DBIndex far *ndx);
extern int pascal far DBIndexes_UpdateUnique( DBFile far *alias,
    DBIndex far *ndx);
#endif

```

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.

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.

```

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 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; buffersize: CARDINAL; safety:BOOLEAN);  
PROCEDURE OpenIndex( VAR ndx: DBIndex):BOOLEAN;  
PROCEDURE CloseIndex(VAR 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.

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

like FindPositionCh, but indexes numeric values rather than string values.

FindPositionCh indexes all fields except numeric 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; buffersize: CARDINAL; safety:BOOLEAN);  
PROCEDURE OpenIndex( VAR ndx: DBIndex):BOOLEAN;  
PROCEDURE CloseIndex(VAR 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.

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.

```
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.

```
PROCEDURE CurrentKeyCh(VAR ndx: DBIndex;VAR val:ARRAY OF CHAR);  
PROCEDURE CurrentKeyN(VAR ndx: DBIndex):Real8;  
PROCEDURE CurrentRec(VAR ndx: DBIndex):LONGINT;  
PROCEDURE KeyType(VAR ndx: DBIndex):CHAR;  
PROCEDURE KeyLength(VAR 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 InsertEntry(VAR ndx: DBIndex; kstr: ARRAY  
    OF CHAR; recno: LONGINT);
```

Insert entry is needed for adding keys that are not a simple field. For simple fields, AddRecord does everything for you automatically. InsertEntry's function is simply to insert kstr at the current position in the index. It will probably cease to be exported when handling of indexes based on keyprocedures is improved.

The following is a code segment using InsertEntry with a Complex indexkey.


```

PROCEDURE KeyByDateAEProcedure
( 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 KeyByDateAEProcedure;

```

```

IF Scrn2DBF.FrameToDBF( Apointment,
  DspFiles.MainFrame, dumcard) THEN
END (* if FrameToDBF *);
ModBase3.WriteDBRec( Apointment );
KeyByDateAEProcedure( Apointment, str );
DBIndexes.FindPositionCh( ApointmentByDateAE, str,
  dumbbool );
DBIndexes.InsertEntry( ApointmentByDateAE, str,
  Apointment.currentrecnum );

```

```

PROCEDURE BuildIndex(VAR ndx: DBIndex; VAR alias:
  DBFile; keyexp: CARDINAL; name: ARRAY OF CHAR);

```

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.

```

PROCEDURE InitCompIndex(indexname: ARRAY OF CHAR; VAR
  ndx: DBIndex; Key: KeyProcedure; buffersize:
  CARDINAL ; safety: BOOLEAN);
PROCEDURE BuildCompIndex( VAR ndx: DBIndex; type:
  CHAR; VAR alias: DBFile; keyexp, size :
  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 must be 'C' in the present release. The keyexp parameter is not used. 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 a good 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, OK);
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 rebalance 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 similar to UpdateIndex, but its purpose is to let you control whether duplicate entries will be written to an index. If you choose to use it, you must call it after making changes to a record before calling WriteDBFile. It will return TRUE if the key is not changed or if the new value of the key is not in the index. It will return FALSE only if the Update will produce a duplicate.

1.8 DBList

```
(*Start of JPI directives :*)
(*$D DATA*)
(*$M ModBase*)
(*End of JPI directives*)
```

```
DEFINITION MODULE DBList;
```

```
(*
 *   ModBase
 *   Release 2.1
 *   By Don Fletcher & John McMonagle
 *   (c) Copyright 1986, 1987, 1988 PMI
 *   4536 SE 50th
 *   Portland, Oregon 97206
 *   All Rights Reserved
 *)
```

```
FROM ModBase3 IMPORT DBFile;
```

```
PROCEDURE List (VAR alias: DBFile; scope: CARDINAL;
                fields: ARRAY OF CHAR);
```

```
PROCEDURE GetNamedField(alias: DBFile; fldname: ARRAY
                        OF CHAR; VAR field: ARRAY OF CHAR): BOOLEAN;
```

```
END DBList.
```

C Header File:

```
#ifndef DBList_Def
#define DBList_Def

void far pascal DBList();

#include "ModBase3.H"

extern void pascal far DBList_List( DBFile far *alias,
    unsigned scope, unsigned High_fields, char far *fields);

extern int pascal far DBList_GetNamedField( DBFile alias,
    unsigned High_fldname, char far *fldname,
    unsigned High_field, char far *field);

#endif
```

DBList is a dBase-like listing facility, demonstrating a method of passing several field names to a list procedure in a comma-delimited string, instead of by field number.


```
PROCEDURE List (VAR alias: DBFile; scope: CARDINAL;  
fields: ARRAY OF CHAR);
```

List lists a Dbase3 file to the standard output device (i.e., the screen, unless output has been redirected) in somewhat the same manner as DBase3's List function. It lists the named fields for the current record and subsequent records as specified by scope. The 'fields' parameter should name valid fields separated by commas, e.g., "NUMBER, DATE, AMOUNT, PAYEE".

```
PROCEDURE GetNamedField(alias: DBFile; fldname: ARRAY  
OF CHAR; VAR field: ARRAY OF CHAR): BOOLEAN;
```

GetNamedField retrieves a field from the current record of 'alias' with the field specified by name rather than number. It is than using the DBField module's PosOfField procedure, which provides a better general solution.

1.9 DBRelate

```
(*Start of JPI directives :*)
(*$D DATA*)
(*$M ModBase*)
(*End of JPI directives*)
```

```
DEFINITION MODULE DBRelate;
```

```
( *
 *   ModBase
 *   Release 2.1
 *   By Don Fletcher & John McMonagle
 *   (c) Copyright 1986, 1987, 1988 PMI
 *   4536 SE 50th
 *   Portland, Oregon 97206
 *   All Rights Reserved
 *
 *)
```

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

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

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

```
END DBRelate.
```

C Header File:

```
#ifndef DBRelate_Def
#define DBRelate_Def
```

```
void far pascal DBRelate();
```

```
#include "ModBase3.H"
#include "DBIndexes.H"
```

```
extern int pascal far DBRelate_ReadRelate(
    unsigned High_key, char far *key, DBIndex far *ndx,
    DBFile far *dbf);
```

```
extern unsigned pascal far DBRelate_PosOfField( DBFile far *alias,
    unsigned High_fieldname, char far *fieldname);
#endif
```

```
PROCEDURE ReadRelate( key: ARRAY OF CHAR; VAR ndx:
    DBIndex; VAR 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 in stead of one:

Customer file fields

Name
StreetAddress
ZipCode

ZipFile Fields

ZipCode
City
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.

```
PROCEDURE PosOfField(VAR 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.

1.10 DBScreen

```
(*Start of JPI directives :*)
(*$D DATA*)
(*$M ModBase*)
(*End of JPI directives*)
```

```
DEFINITION MODULE DBScreen;
```

```
(*
 *   ModBase
 *   Release 2.1
 *   By Don Fletcher & John McMonagle
 *   (c) Copyright 1986, 1987, 1988 PMI
 *   4536 SE 50th
 *   Portland, Oregon 97206
 *   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.
```


C Header File:

```
#ifndef DBScreen_Def
#define DBScreen_Def

void far pascal DBScreen();

#include "SmartScreen.H"
#include "ModBase3.H"

typedef enum {
    Ahead,
    Nowhere,
    Back,
    Escape
} Direction;

struct {
    Colors fore, back;
    TextAttribute MonoAttr;
extern } DBScreen_readattr, DBScreen_writeattr;
extern Direction DBScreen_lastdirection;

extern void pascal far DBScreen_CreateDBF(
    unsigned High_dbfilename, char far *dbfilename,
    DBFile far *alias);

extern void pascal far DBScreen_GetDescriptors(
    unsigned High_fields, DBFieldDescriptor far *fields);

extern void pascal far DBScreen_ModifyDBStruc( DBFile far *alias);

extern void pascal far DBScreen_Say( unsigned row, unsigned col,
    unsigned High_s, char far *s);

extern void pascal far DBScreen_Get( unsigned row, unsigned col,
    unsigned High_s, char far *s);

extern void pascal far DBScreen_ReadGets( void);

extern unsigned pascal far DBScreen_Row( void);

extern unsigned pascal far DBScreen_Col( void);

#endif
```

DBScreen uses Repertoire's SmartScreen and WindowPrims modules to implement a simple form of input handling that resembles the one used by dBasIII. Because the procedures are implemented through Repertoire's 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: ');  
Get(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 ReverseVideo 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.

1.11 MemoFunctions

```
(*Start of JPI directives :*)
(*$D DATA*)
(*$M ModBase*)
(*End of JPI directives*)

DEFINITION MODULE MemoFunctions;
(*
 * ModBase
 * Release 2.1
 * By Don Fletcher & John McMonagle
 * (c) Copyright 1986, 1987, 1988 PMI
 * 4536 SE 50th
 * Portland, Oregon 97206
 * 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.
```

C Header File:

```
#ifndef MemoFunctions_Def
#define MemoFunctions_Def

void far pascal MemoFunctions();

#include "ModBase3.H"

extern void pascal far MemoFunctions_GetMemoField( DBFile far *ali
    unsigned fieldnum, unsigned High_memo, char far *memo);

extern void pascal far MemoFunctions_ReplaceM( DBFile far *alias,
    unsigned fieldnum, unsigned High_memo, char far *memo);
#endif
```

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:

```
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.

1.12 MiscFunctions

```
(*Start of JPI directives :*)
(*$D DATA*)
(*$M ModBase*)
(*End of JPI directives*)
```

DEFINITION MODULE MiscFunctions;

```
(*
 *   ModBase
 *   Release 2.1
 *   By Don Fletcher & John McMonagle
 *   (c) Copyright 1986, 1987, 1988 PMI
 *   4536 SE 50th
 *   Portland, Oregon 97206
 *   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.
```

C Header File:

```
#ifndef MiscFunctions_Def
#define MiscFunctions_Def

void far pascal MiscFunctions();

extern int pascal far MiscFunctions_Alph( char c);
```



```

extern int pascal far MiscFunctions_Digit( char c);
extern int pascal far MiscFunctions_AlphaNum( char c);
extern int pascal far MiscFunctions_Punc( char c);
extern int pascal far MiscFunctions_Printable( char c);
extern int pascal far MiscFunctions_IsLower( char c);
extern int pascal far MiscFunctions_IsUpper( char c);
extern int pascal far MiscFunctions_IsSpace( char c);

extern void pascal far MiscFunctions_Trim(
    unsigned High_str, char far *str);

extern void pascal far MiscFunctions_Upper(
    unsigned High_str, char far *str);

extern int pascal far MiscFunctions_FieldName(
    unsigned High_str, char far *str);

extern int pascal far MiscFunctions_FieldNameChar( char c);
#endif

```

Alpha 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.

1.13 ModBase3

```
(*Start of JPI directives :*)
(*$D DATA*)
(*$M ModBase*)
(*End of JPI directives*)
DEFINITION MODULE ModBase3;
(*
 * ModBase
 * Release 2.1
 * By Don Fletcher & John McMonagle
 * (c) Copyright 1986, 1987, 1988 PMI
 * 4536 SE 50th
 * Portland, Oregon 97206
 * All Rights Reserved
 *
 *)
```

```
FROM SYSTEM IMPORT
    ADDRESS;
```

CONST

```
MaxRecLength = 4000;
MaxField = 128;
MaxHeaderLen = 32*(MaxField+1)+2;
NameLen = 30;
MemoSize = 512;
```

TYPE

```
DBFieldDescriptor =
    RECORD
        name: ARRAY [0..9] OF CHAR;
        size: CARDINAL;
        decplaces: CARDINAL;
        offset: CARDINAL;
        fldtype: CHAR;
    END;
```

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

```
DBFile =
    RECORD
        fileID,
        MemoHandle: CARDINAL;
        open,
        MemoOpen,
        (* will open only if accessed *)
        Safety: BOOLEAN;
        (* if true keeps disk up to date open sets
           to true *)
```



```

    appending,
    readingBuffer,
    hasmemo: BOOLEAN;
    Init,
    numberoffields: CARDINAL;
    lastupdate: ARRAY[0..2] OF CARDINAL;
    length: CARDINAL; (* of records in BYTES *)
    fieldlist: POINTER TO DBFieldArray;
    numofrecords: LONGINT;
    currentrecnum: LONGINT;
    headerlength: CARDINAL;
    SavePtr,currentrec,BufferPtr: POINTER TO ARRAY
        [1..MaxRecLength] OF CHAR;
    name,
    MemoName: ARRAY [0..NameLen] OF CHAR;
    dbbuffer : ADDRESS;
    buffersize, (* requested size *)
    size : CARDINAL; (* size of buffer in bytes *)
    start : LONGINT; (* first record number *)
    MaxRecords, NumRecords : CARDINAL;
    IndexList:ADDRESS;
END;

IndexCheckProcedure=PROCEDURE(VAR DBFile);

VAR
UpdateIndexes:IndexCheckProcedure;

PROCEDURE InitDBF(filename: ARRAY OF CHAR; VAR alias:
    DBFile; BufferSize:CARDINAL; safety:BOOLEAN );

PROCEDURE OpenDBF
    ( VAR alias      : DBFile ):BOOLEAN;

PROCEDURE CloseDBF(VAR alias: DBFile);

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

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

PROCEDURE SetDBBuffer(VAR alias: DBFile; BufferSize:
    CARDINAL );
    (* BufferSize in bytes; will set to multiple of
    alias.length *)

PROCEDURE WriteDBRec(VAR alias: DBFile);

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

PROCEDURE AppendBlank(VAR alias: DBFile);

```



```

PROCEDURE DeleteRecord(VAR alias: DBFile);
PROCEDURE UnDeleteRecord(VAR alias: DBFile);
PROCEDURE Deleted(VAR alias: DBFile): BOOLEAN;
PROCEDURE SetDBSafetyOn(VAR alias: DBFile);
PROCEDURE SetDBSafetyOff(VAR alias: DBFile);
PROCEDURE SetCurrToDBBuffer(VAR alias: DBFile);
PROCEDURE SetCurrToDBCurr(VAR alias: DBFile);

END ModBase3.

```

C Header File:

```

#ifndef ModBase3_Def
#define ModBase3_Def

void far pascal ModBase3();

#define MaxRecLength 4000
#define MaxField 128
#define MaxHeaderLen 32*(MaxField+1)+2
#define NameLen 30
#define MemoSize 512

typedef
struct DBFieldDescriptor {
    char name[10];
    unsigned size;
    unsigned decplaces;
    unsigned offset;
    char fldtype;
} DBFieldDescriptor;

typedef
DBFieldDescriptor DBFieldArray[MaxField];

typedef
struct DBFile {
    unsigned fileID, MemoHandle;
    int open, MemoOpen, Safety;
    int appending, readingBuffer, hasmemo;
    unsigned Init, numberoffields;
    unsigned lastupdate[3];
    unsigned length;
    DBFieldArray far *fieldlist;
    long numofrecords;
    long currentrecnum;
    unsigned headerlength;
}

```



```

    char far *SavePtr[MaxRecLength], far *currentrec[MaxRecLength],
    far *BufferPtr[MaxRecLength];
    char name[NameLen+1], MemoName[NameLen+1];
    void far *dbbuffer;
    unsigned buffersize, size;
    long start;
    unsigned MaxRecords, NumRecords;
    void far *IndexList;
} DBFile;

typedef void (far * pascal IndexCheckProcedure) (DBFile far *);

extern IndexCheckProcedure ModBase3_UpdateIndexes;

extern void pascal far ModBase3_InitDBF(
    unsigned High_filename, char far *filename, DBFile far *alias,
    unsigned BufferSize, int safety);

extern int pascal far ModBase3_OpenDBF( DBFile far *alias);

extern void pascal far ModBase3_CloseDBF( DBFile far *alias);

extern void pascal far ModBase3_UpdatedBFile( DBFile far *alias);

extern void pascal far ModBase3_ReadDBRec( DBFile far *alias,
    long recnum);

extern void pascal far ModBase3_SetDBBuffer( DBFile far *alias,
    unsigned BufferSize);

extern void pascal far ModBase3_WriteDBRec( DBFile far *alias);

extern void pascal far ModBase3_GetField( DBFile far *alias,
    unsigned fieldnumber, unsigned High_field, char far *field,
    int far *ok);

extern void pascal far ModBase3_AppendBlank( DBFile far *alias);

extern void pascal far ModBase3_DeleteRecord( DBFile far *alias);

extern void pascal far ModBase3_UnDeleteRecord( DBFile far *alias);

extern int pascal far ModBase3_Deleted( DBFile far *alias);

extern void pascal far ModBase3_SetDBSafetyOn( DBFile far *alias);

extern void pascal far ModBase3_SetDBSafetyOff( DBFile far *alias);

extern void pascal far ModBase3_SetCurrToDBBuffer( DBFile far *alias);

extern void pascal far ModBase3_SetCurrToDBCurr( DBFile far *alias);
#endif

```


ModBase3 exports a record type called DBFile which 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.

Most of the fields of the DBFile type are meant only for ModBase's internal use. Some of them, however, you will have to access directly. The numofrecords LONGINT, for example, lets you write loops that process all the records in a data file by simply counting from 1 to the last record. Similarly, the numberoffields CARDINAL lets you process all the fields in a record. Note that in 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.

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

```
PROCEDURE InitDBF(filename: ARRAY OF CHAR; VAR alias:
    DBFile; BufferSize: CARDINAL; safety: BOOLEAN );
PROCEDURE OpenDBF(VAR alias: DBFile): BOOLEAN;
PROCEDURE CloseDBF(VAR alias: DBFile);
```

In ModBase's 2.1 release, 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 to release memory allocated by InitDBF and OpenDBF.

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.

OpenDBF opens a disk file specified by the name associated with the datafile 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 into its currentrec field. If the operation is unsuccessful for any reason, alias.open is set to FALSE. Note that the filename need not use the '.DBF' filename extension required by dBase. CloseDBF closes an opened dBase file and sets alias.open to FALSE.

DBFile is not a hidden type, so any program has access to any of its fields. This allows the programmer greater control, but carries with it the danger that some value might be inadvertently, or incorrectly, changed. For instance, the BOOLEAN variable open might be assigned the value TRUE when, in fact, the data file is closed. Most of the variables in DBFile should be considered read-only. The main exception to this stricture is currentrec, which is the programmer's window into the data file. The value of the record indicated by currentrecnum is held by currentrec.

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


ReadDBRec reads the "recnum-th" record and stores it in alias.currentrec. If the file identified by alias is not open, or if recnum exceeds alias.numofrecords (i.e., the number of records in the file), WriteDBRec writes currentrec to disk. No changes made to currentrec will be written to the data file unless WriteDBRec is called. A feature inherited from dBase is that the first character in currentrec 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(VAR alias: DBFile);
PROCEDURE UnDeleteRecord(VAR alias: DBFile);
PROCEDURE Deleted(VAR 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( VAR alias: DBFile; fieldnumber:
    CARDINAL; VAR field: ARRAY OF CHAR; VAR ok:
    BOOLEAN);
```

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 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.

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

AppendBlank is the procedure that adds new records to the data file. It appends a blank record to the file, updates the numofrecords variable in the referenced DBFile, and puts a blank string in the currentrec variable. AppendBlank makes the added record the current record.

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

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

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


UpdateDBFile has the effect of closing and opening the dbfile but is faster. Used when safety is off to update DBFile and Memos to disk. Safety is by default on. It has a significant speed penalty on writes, particularly appends, 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.

The procedures SetCurrToDBBuffer and SetCurrToDBCurr are meant only management of the read/write buffer that ModBase uses for performance purposes. You should never have to use them directly.

1.14 Scrn2DBF

```
(*Start of JPI directives :*)
(*$D DATA*)
(*$M ModBase*)
(*End of JPI directives*)

DEFINITION MODULE Scrn2DBF;
(*)
*   ModBase
*   Release 2.1
*   By Don Fletcher & John McMonagle
*   (c) Copyright 1986, 1987, 1988 PMI
*   4536 SE 50th
*   Portland, Oregon 97206
*   All Rights Reserved
*
*)

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.
```

C Header File:

```
#ifndef Scrn2DBF_Def
#define Scrn2DBF_Def

void far pascal Scrn2DBF();

#include "ModBase3.H"
#include "ScrnTypes.H"

extern int pascal far Scrn2DBF_FrameToDBF( DBFile far *DBF,
  DisplayFrame far *TheFrame);

extern void pascal far Scrn2DBF_DBFToFrame( DBFile far *DBF,
  DisplayFrame far *TheFrame);
#endif
```

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 modules 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 Release 2.0's generally transparent handling of memo files, Scm2DBF 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 wraps 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.

1.15 Definition Module Compilation Order.

If you find it necessary to recompile ModBase's definition modules, do so in the following order:

```
ModBase3.DEF .  
DBIndexes.DEF .  
ChkInd.DEF .  
DBCopier.DEF .  
DBCreate.DEF .  
MiscFunc.DEF .  
DBFields.DEF .  
MemoFunc.DEF .  
DBList.DEF .  
DBRelate.DEF .  
DBScreen.DEF .  
Scrn2DBF.DEF
```

Implementation modules can be recompiled in any order if definition modules are recompiled in the proper order. We strongly recommend, however, that you use your compiler's Make utility instead of recompiling all of ModBase's .DEFs. Modula-2 Make utilities are smarter than Unix/C Make utilities -- they read your source code, so you do not have to build and maintain a Make file. If you don't know how to use your Make utility, stop now and spend the five minutes that it takes to learn. You will never have to worry about definition module compilation order again.

Index

AddRecord:	19	KeyType:	19
AddToUpdateList:	19	List:	26
Alph:	36	MaxField:	40
AlphaNum:	36	MaxHeaderLen:	40
AppendBlank:	40	MaxRecLength:	40
BuildCompIndex:	19	MemoSize:	40
BuildDBF:	13	ModifyDBStruc:	30
BuildIndex:	19	MonoAttr:	30
CloseDBF:	40	NameLen:	40
CloseIndex:	19	NDXChk:	10
Col:	30	NextRecord:	19
CreateDBF:	30	OpenDBF:	40
CurrentKeyCh:	19	OpenIndex:	19
CurrentKeyN:	19	PosOfField:	28
CurrentRec:	19	PrevRecord:	19
DataFormatToDate:	15	Printable:	36
DateToDataFormat:	15	Punc:	36
DBCpy:	11	readattr:	30
DBFieldCopy:	11	ReadDBRec:	40
DBFieldDescriptor:	40	ReadGets:	30
DBFToFrame:	45	ReadRelate:	28
DBPack:	11	Replace:	15
DeleteCurrentEntry:	19	Replaced:	15
Deleted:	40	ReplaceL:	15
DeleteRecord:	40	ReplaceM:	33
Digit:	36	ReplaceN:	15
Direction:	30	Row:	30
DisposeIndex:	19	Say:	30
FieldName:	36	SetCurrToDBCurr:	40
FieldNameChar:	36	SetCurrToDBBuffer:	40
FindPositionN:	19	SetDBBuffer:	40
FindPositionCh:	19	SetDBSafetyOn:	40
FrameToDBF:	45	SetDBSafetyOff:	40
Get:	30	SetIndexBuffers:	19
GetDateField:	15	SetSafetyOff:	19
GetDescriptors:	30	SetSafetyOn:	19
GetField:	40	Trim:	36
GetLogicalField:	15	UnDeleteRecord:	40
GetMemoField:	33	UpdateDBFile:	40
GetNamedField:	26	UpdateIndexes:	40
GetNumField:	15	UpdateIndex:	19
GoBottom:	19	UpdateUnique:	19
GoTop:	19	Upper:	36
InitCompIndex:	19	writeattr:	30
InitDBF:	40	WriteDBRec:	40
InitIndex:	19		
InsertEntry:	19		
IsLower:	36		
IsSpace:	36		
IsUpper:	36		
KeyLength:	19		
KeyProcedure:	19		



POLYTRON Corporation, 1700 NW 167th Place, Beaverton, OR 97006 USA
(503) 645-1150, FAX (503) 645-4576, TELEX 325800 POLYTRON